Ong Yeow Hwee, Glen

Singapore University of Social Science

ICT337: Big Data Computing in the Cloud

Prof. Hu He

October 19, 2023

**Question 1**

**(a)**

**Premise**

Both PySpark DataFrame and PySpark Resilient Distributed Datasets (RDD) are abstractions offered by Apache Spark for distributed data processing. They act as APIs that give Spark framework users the ability to interact with and manipulate data. These two fundamental components of Spark provide diverse methods for distributed data processing, each with unique advantages and applications. In this project, I will discuss the concepts of PySpark RDD and PySpark DataFrame by exploring their respective characteristics, transformations, actions and use cases.

***Resilient Distributed Datasets (RDD)***

**Definition and Characteristics.** The main data structure used by Apache Spark is RDD. It is a set of immutable, fault-tolerant components that can be handled concurrently by a cluster of machines. RDDs make the physical properties of the data, such as partitions, accessible to users, allowing them to understand how the data is processed and distributed in a distributed computing environment. RDDs also use lineage graph to keep track of the data transformations performed, ensuring resiliency and the capacity to recompute missing or damaged partitions in the event of node failures.

**Transformations and Actions.** When performing transformations, a new RDD is produced by applying a function to the elements of an existing RDD. Common transformations such as "map" and "filter" follow the lazy evaluation paradigm, which means they postpone execution and create a chain of transformations. This strategy enables Spark to effectively optimize the execution plan. The crucial points where Spark executes the computations are represented by actions. Actions like "collect" and "reduce" trigger the previously defined lineage of transformations to be evaluated, which computes the desired result.

**Use Case.** RDDs are best suited for situations in which precise, low-level control over data processing is essential. They excel in scenarios involving semi-structed or unstructured data sources. In situations requiring specialized data manipulation and the execution of challenging, non-standard data processing workflows, RDDs truly shine. They are a versatile option for developers taking on complex data processing challenges due to their flexibility and capacity to handle complex data operations.

***DataFrames***

**Definition and Characteristics.** As a higher-level abstraction built on top of RDDs, DataFrames in PySpark handles structured data effectively. A DataFrame is essentially a distributed collection of named columns and rows that is immutable and distributed across a cluster of machines. The ability of Spark to automatically understand the data's structure and optimize query execution for better performance is one of the key benefits of DataFrames. DataFrames’ user-friendly and structured approach to data manipulation is especially well-suited for a variety of data processing tasks, especially those involving structured data sources.

**Transformations and Actions.** DataFrames provide a more extensive array of high-level operations, enhancing their usability. Transformation operations such as “select” and “groupBy” enable users to manipulate data with ease while adhering to a lazy evaluation paradigm, which postpones execution and creates a chain of transformations. On the other hand, action operations like “count” and “show” serve as triggers, initiating the execution of transformations and returning valuable insights. This combination of rich high-level operations and efficient execution mechanisms makes DataFrames an appealing choice for various data processing tasks, particularly those involving structured data sources.

**Use Case.** When it comes to processing structured data, DataFrames becomes the favored solution for a variety of data processing tasks. They offer streamlined operations for data exploration and wrangling in scenarios involving structured data formats like CSV, Parquet, or Avro, where their effectiveness really shines. The unique feature of DataFrames is their built-in compatibility with SQL-like queries, which gives users a simple way to query their data. Additionally, the Catalyst optimizer's inclusion guarantees automatic optimization of query execution plans, improving performance and making DataFrames particularly advantageous for tasks that require both structured data processing and query optimization.

Abstraction level Low-level, more control over the pre-processing part High-level, rich semantics

**(b)**

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Resilient Distributed Datasets (RDD)** | **DataFrames** |
| Abstraction level | Low-level, more control over pre-processing | High-level, rich semantics |
| Data structure | Unstructured | Structured (tabular) |
| Schema | No schema | Schema with named columns |
| Typing | No type enforcement | Strong typed |
| Catalyst optimizer | Not applicable | Optimizes query execution plans |
| Serialization | User-defined | Built-in |
| Fault-tolerance | Basic (requires lineage) | Built-in |
| Spark SQL | No | Yes |
| Transformations | ‘map’, ‘reduce’, custom transformations | ‘select’, ‘filter’, SQL queries, joins |
| Ease of use | More complex | More user-friendly |

*Table 1.1. Differences between PySpark RDD and PySpark Dataframes.*

**Question 2**

**(a)**

***Premise***

My local machine operates in windows, it is advantageous to create an isolated environment for running Apache Spark. The VMware Workstation 17 Player is downloaded, installed with the ubuntu iso, and launched (see Figure 1 and Figure 2). Subsequently, the following processes are applied within the VMware environment.

A screenshot of a computer

Description automatically generated

*Figure 1. Creating a Virtual machine in VMware with Ubuntu iso image.*

A screenshot of a computer

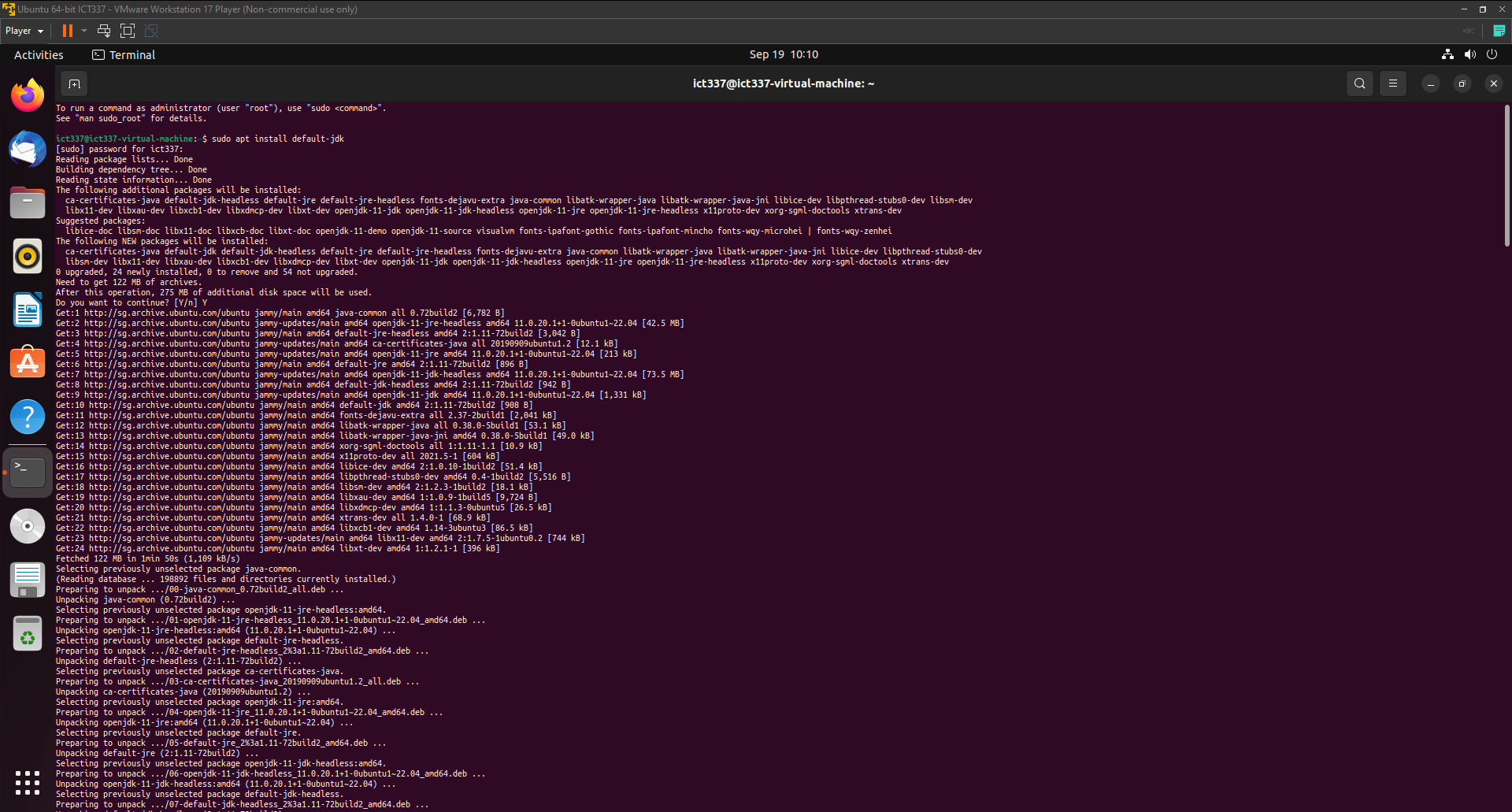
Description automatically generated

*Figure 2. Virtual machine configuration in VMware.*

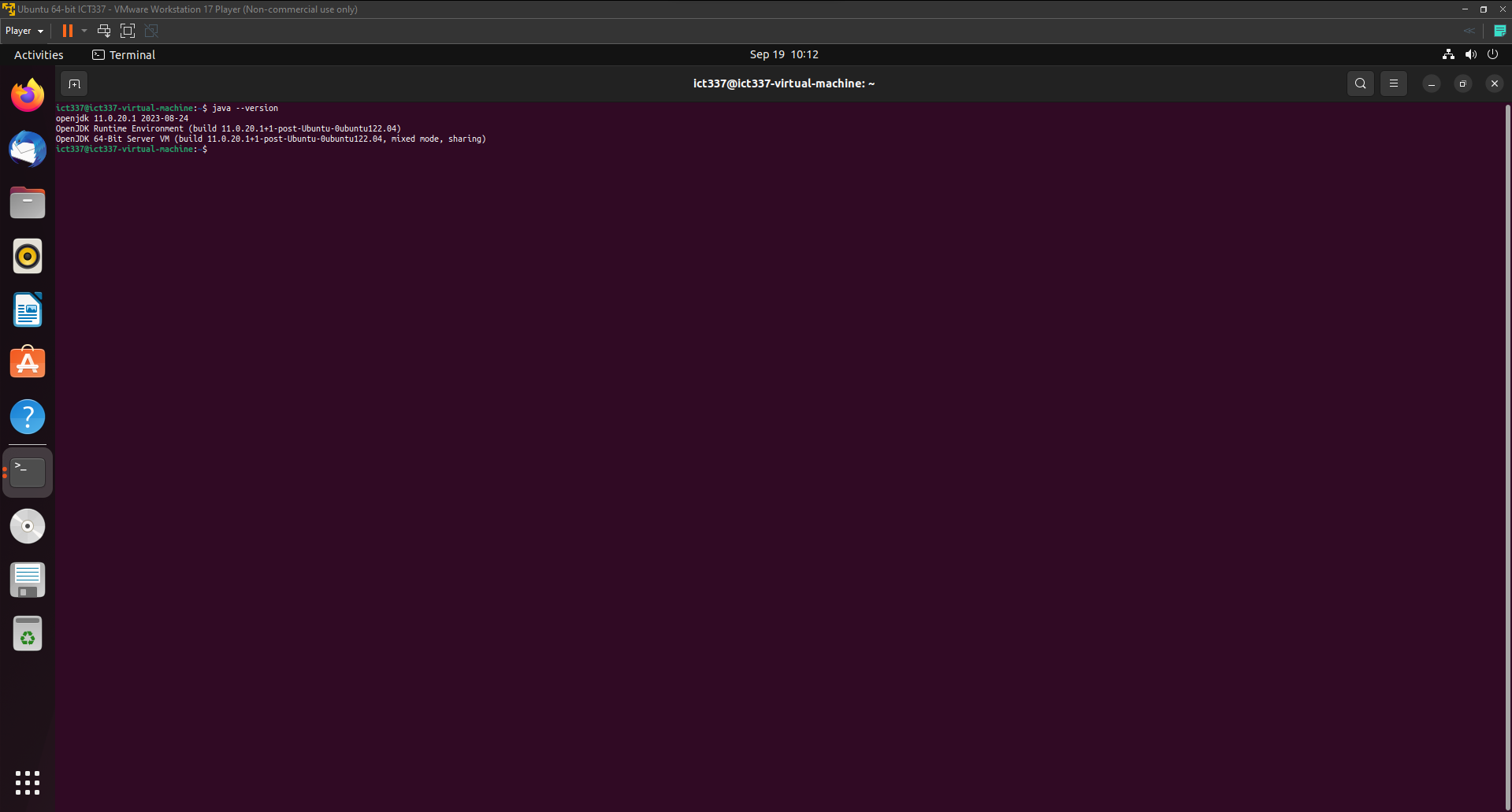
***Prerequisites***

Apache Spark requires Java Development Kit (1) and Python (2). In my case, I chose to download Apache Spark version 3.4.1 and the pre-built for Apache Hadoop package type from Apache Sparks official website (3).

Firstly, ‘sudo apt install default-jdk’ command is used in the terminal to download the Java Development Kit prerequisite and the ‘java –version’ command is used to verify if the Java has been installed accordingly (see Figure 3 and Figure 4).

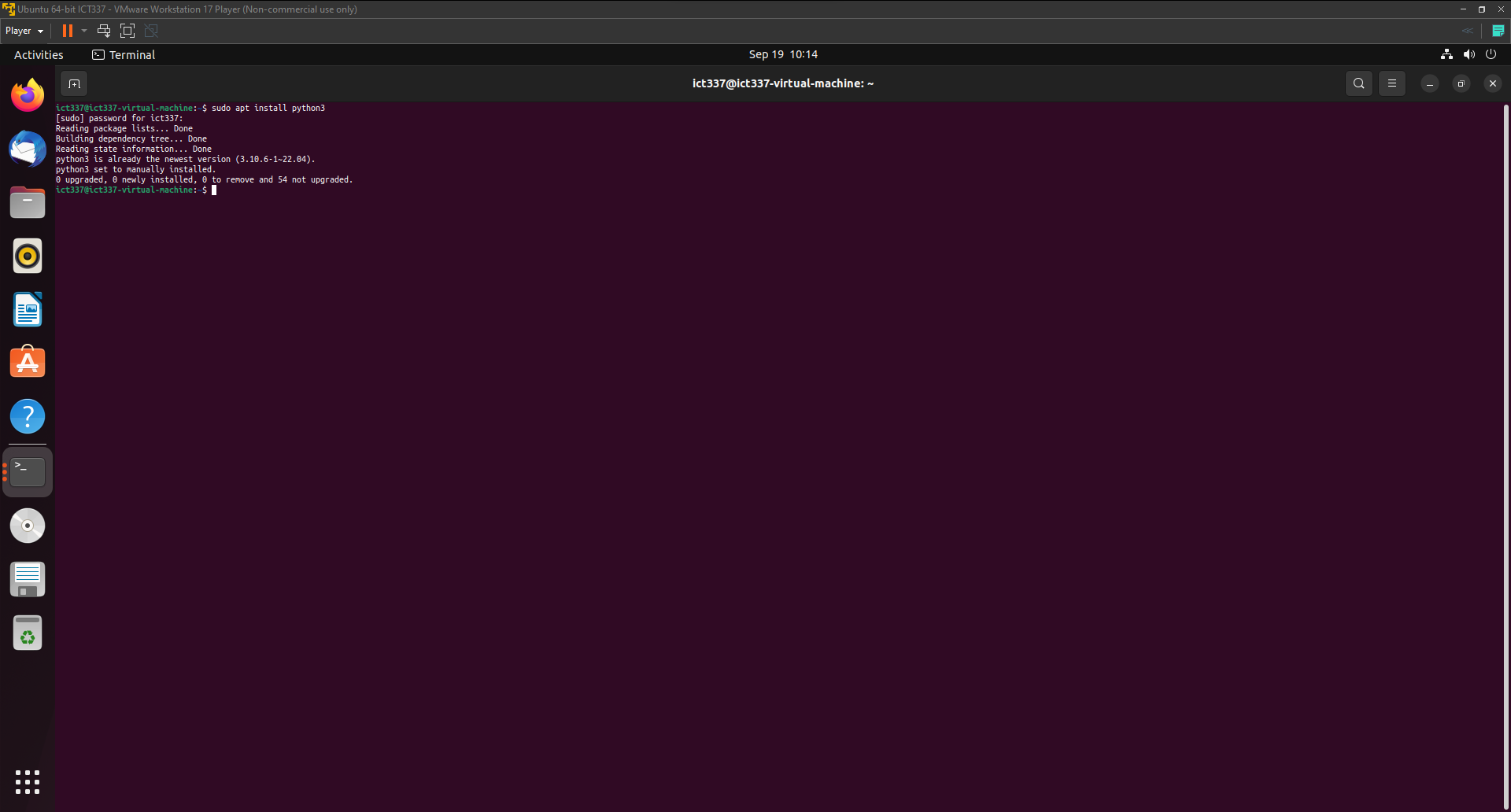


*Figure 3. Sample Java Development Kit installation in Ubuntu terminal.*



*Figure 4. Java Development Kit installation verification in Ubuntu terminal.*

Secondly, ‘sudo apt install python3’ command is used in the terminal to download the Python prerequisite and the ‘python3 –version’ command is used to verify if the Python has been installed accordingly (see Figure 5 and Figure 6).



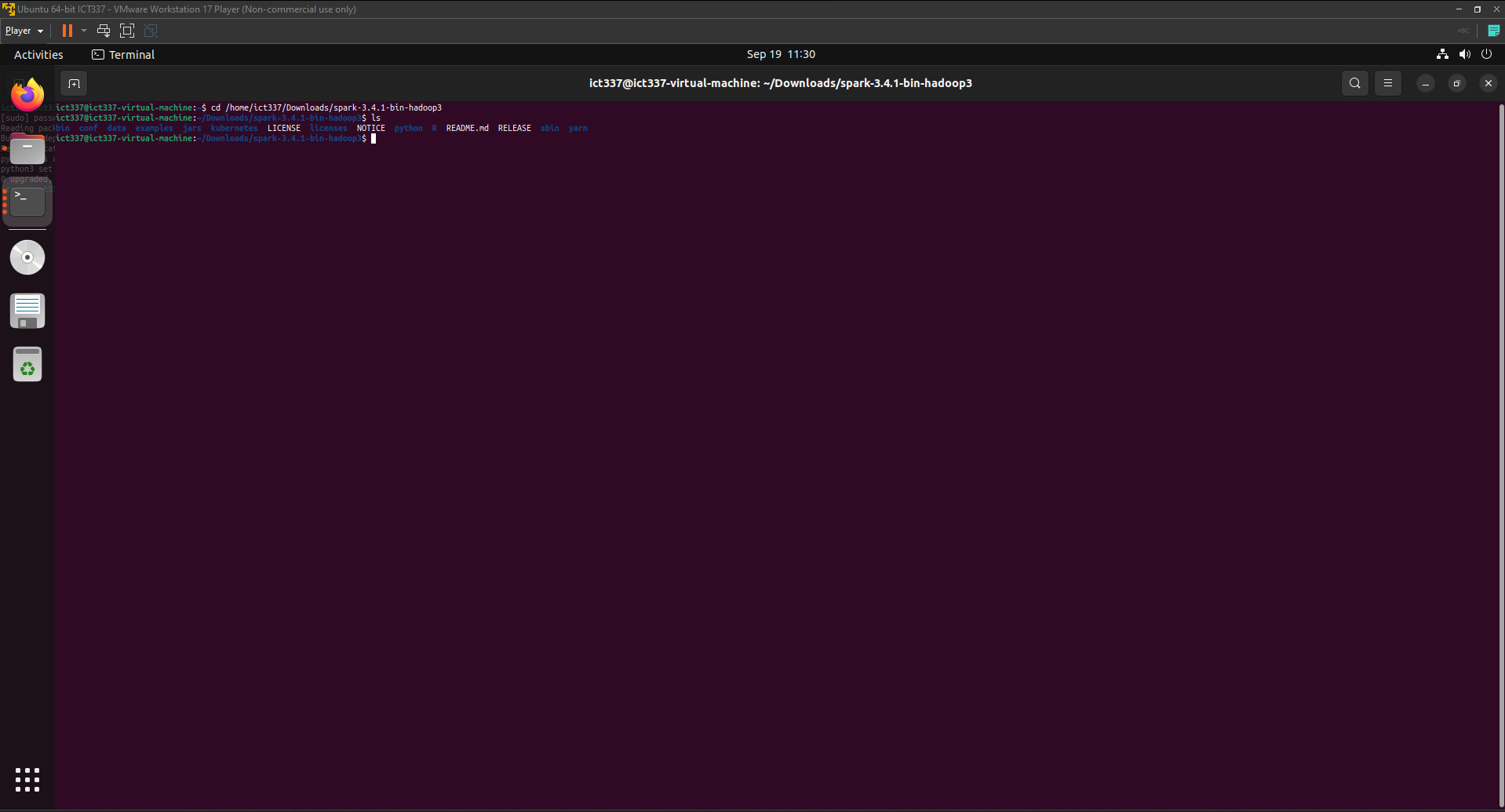
*Figure 5. Python3 installation in Ubuntu terminal.*

A screenshot of a computer

Description automatically generated

*Figure 6. Python3 installation verification in Ubuntu terminal.*

Thirdly, the Spark 3.4.1 release with pre-built for Apache Hadoop 3.3 and later is acquired from the Apache Spark official website. After obtaining the release, the.tgz file contents are extracted. In the terminal, ‘cd /home/ict337/Downloads/spark-3.4.1-bin-hadoop3’ command is used to navigate into the Spark Hadoop package directory and the ‘ls’ command is used to inspect the Hadoop folder (See Figure 7). To ensure the Spark Hadoop package is in place and functioning correctly, the ‘spark-shell’ is executed and Spark Web User Interfaced is launched. To accomplish this, the ‘cd bin’ command is used to navigate into the bin folder, followed by the execution of the ‘./spark-shell’ command (See Figure 8 and Figure 9).



*Figure 7. Changing directory to the downloaded and extracted .tgz file in Ubuntu terminal.*

A screenshot of a computer

Description automatically generated

*Figure 8. Execution of spark-shell in Ubuntu terminal*

A screenshot of a computer

Description automatically generated

*Figure 9. Spark Web User Interface*

***Demonstration***

To demonstrate the non-interactive execution of built-in ‘pi.py’ example program, a new terminal window is opened and navigated into the into the Spark Hadoop package directory with the ‘cd /home/ict337/Downloads/spark-3.4.1-bin-hadoop3’ (See Figure 10). The ‘pi.py’ example program provided leverages Monte Carlo method to estimate pi. Notably, the number of partitions employed for the estimation is defaulted to 2 when no partition count is supplied in the command-line argument. This default behavior is governed by ‘partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2’ line of code in the pi.py example program. Therefore, the ‘./bin/spark-submit ./examples/src/main/python/pi.py’ command would suffice, returning a pi estimation of 3.135800 (see Figure 11).

A screenshot of a computer

Description automatically generated

*Figure 10. Changing directory to the Spark Hadoop package file in Ubuntu terminal.*

A screenshot of a computer screen

Description automatically generated

*Figure 11. Execution of ‘pi.py’ program without partitions supplied*

**(b)**

***Dependencies***

It is likely that the the 'pi.py' example program is written in Python 2 and to ensure compatibility with Python 2 and Python 3, the 'from \_future\_ import print\_function' statement is used to leverage the Python 3-style 'print' function such as '%f' formatter in Python 2. The 'import sys' line of code imports the 'sys' module to provide access to system-specific parameters. In the 'pi.py' example program, it uses command line arguments such as the 'sys.argv[]' and 'len(sys.argv)'.

Additionally, the 'from random import random' line of code imports the random function from the random module and the 'from operator import add' line of code imports the add function from the operator module. These functions are leveraged in 'pi.py' example program for randomizing the x and y axis and counting.

Furthermore, the 'from pyspark.sql import SparkSession' line of code imports the SparkSession class from the pyspark.sql module. This is used in the 'pi.py' example program as an API or entry point for configuring, creating, and interacting with Spark functionalities.

***Initialization***

Firstly, the 'if \_name\_ = "\_\_main\_\_"' line of code checks whether the script is executed as the main program to prevent unintended execution of the 'pi.py' example program. Additionally, the docstring of 'Usage: pi [partitions]' encased in the 3 double quotation marks provides the usage instructions for how to execute the 'pi.py' example program, indicating that it expects a 'partitions' argument to specify the number of partitions.

Secondly, inside the 'if \_name\_ = "\_\_main\_\_":' block of code, the 'SparkSession' class and '.builder' method is called to create and configure a spark instance respectively. Additionally, the '.appName("PythonPi")' method gives the application name of the spark job an identifier of "PythonPi". Furthermore, the '.getOrCreate()' method ensures a SparkSession is available for the 'pi.py' example program by checking whether there is an existing SparkSession and creates one if there is none.

***Main Application Logic***

Prior to the declaration of the ‘f’ function, some initial configuration is performed such as the number of partitions (1) which will be divided for parallel processing and determining the total number of data points across the specified number of partitions (2). Firstly, the line of code ‘partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2’ retrieves the ‘partitions’ value from the command-line. This operation identifies arguments specified by the user in the command-line argument. For example, in the command ‘./bin/spark-submit ./examples/src/main/python/pi.py 7’, it contains more than 1 system argument, satisfying the condition of ‘if len(sys.argv) > 1’. Hence the number 7 identified by ‘sys.argv[1]’ would be converted to an integer and assigns it to the ‘partitions’ variable. However, in the case where no arguments are specified in the command-line argument, the value of ‘len(sys.argv)’ would remain at 1. Therefore, the number of partitions to default at 2, as it does not satisfy the condition of ‘if len(sys.argv) > 1’. Secondly, the line of code ‘n = 100000 \* partitions’ multiplies 100000 to the number of partitions specified to be later distributed across the partitions. For example, in the command ‘./bin/spark-submit ./examples/src/main/python/pi.py 7’, spark would distribute 700000 data points across 7 partitions, with each partition handling a portion of the data points for parallel processing.

In the ‘f’ function, the x-coordinate, y-coordinate (1) are defined and determines whether the data point falls within a circle with a radius of 1 (2). Firstly, the lines of code ‘x = random() \* 2 – 1’ and ‘y = random() \* 2 – 1’ generate a floating number from 0 to 1. It subsequently multiplies the random floating number by 2 and subtracts 1 from it to ensure that the coordinates may fall within the range of -1 to 1. This process assigns random values within the range from -1 to 1 to the x-coordinate and y-coordinate. Secondly, the line of code ‘return 1 if x \*\* 2 + y \*\* 2 <= 1 else 0’ calculates the squared Euclidean distance from the given random x-coordinate and y-coordinate to origin. Subsequently, it categorizes the random data points, ‘1’ indicating the data point is within the range of -1 to 1 and ‘0’ indicating the data point is outside of this specified range.

In the line of code ‘count = spark.sparkContext.Parallelize(range(1, n + 1), partitions).map(f).reduce(add)’, it creates a Resilient Distributed Datasets (RDD) (1), uses spark to parallelize the generation of random data points (2), apply the ‘f’ function to each element in the RDD (3), and apply the ‘add’ function to add up all the ‘1’s and ‘0’s (4). Firstly, ‘spark.sparkContext’ is an entry point for the spark session to access spark operations such as creating a RDD and perform distributed data processing tasks. Secondly, the ‘.Parallelize(range(1, n + 1), partitions)’ method creates a RDD ‘n’ elements with a sequence of 1 to ‘n’, where ‘n’ represents the total number of data points. This process subsequently distributes these elements across the specified number of partitions. Thirdly, the ‘.map(f)’ transformation applies the ‘f’ function to the RDD and transforms each element in the RDD into ‘1’s or ‘0’s. Lastly, the ‘reduce(add)’ action executes the transformation in the RDD and aggregates the elements across all the partitions. Therefore, this process effectively counts the number of data points that fall within a circle with a radius of 1.

In the line of code ‘print("Pi is roughly %f" % (4.0 \* count/n))’, the ‘pi.py’ example program outputs ‘Pi is roughly %f’, where ‘%f’ is a placeholder for floating value computed by ‘%(4.0 \* count/n)’. Therefore, it computes the estimate of pi with ‘%(4.0 \* count/n)’ and outputs "Pi is roughly %f" in a single print statement. Finally, the ‘spark.stop()’ method is used to end the spark session to ensure a clean shutdown of spark instance in the ‘pi.py’ example program.

**Question 3**

**Full PySpark Program**

import logging

import os

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, sum, avg, min, max, when

# Constants

SCRIPTS\_DIR = os.path.abspath(os.path.dirname(\_\_file\_\_))

DATA\_DIR = os.path.join(SCRIPTS\_DIR, "..", "data")

FLIGHTS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "flights\_data\_v2.csv")

PLANES\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "planes\_data\_v2.csv")

LOGGING\_LEVEL = logging.INFO

LOAD\_DATA\_ERROR\_MESSAGE = "An error occurred while loading data: {}"

FILE\_NOT\_FOUND\_MESSAGE = "The specified file does not exist: {}"

def configure\_logging():

    """Configure logging settings and return a logger object.

    Returns

    -------

    logger : object

        Logger object for logging messages.

    Notes

    -----

    This function returns a logger object that may be used to log messages

    inside the program and initializes the logging parameters, including the logging level.

    The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    logging.basicConfig(*level*=LOGGING\_LEVEL)

    return logging.getLogger(\_\_name\_\_)

def create\_spark\_session(*app\_name*="TMA\_Data\_Analysis"):

    """

    Create and return a Spark session.

    Parameters

    ----------

    app\_name : str, optional

        The name of the Spark application, by default "TMA\_Data\_Analysis".

    Returns

    -------

    SparkSession

        The Spark session object.

    Notes

    -----

    This function initializes a Spark session, which is the entry point for working with Spark functionality.

    """

    return SparkSession.builder.appName(app\_name)\

        .config("spark.some.config.option", "some-value")\

        .getOrCreate()

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    """

    Show rows of a DataFrame with the option to limit the number of rows displayed.

    Parameters

    ----------

    df : DataFrame

        The DataFrame to be displayed.

    max\_rows : int, optional

        The maximum number of rows to display. Default is 100.

    show\_rows : int, optional

        The rows to display if records is above max rows. Default is 20.

    Returns

    -------

    None

    Notes

    -----

    This function shows the rows from the DataFrame input.

    The DataFrame will only display the first "show\_rows" rows if there are more rows than the specified "max\_rows".

    The DataFrame will display all available rows without truncation if the number of rows is less than "max\_rows".

    """

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def load\_data(*spark*, *logger*, *file\_path*):

    """

    Load data from CSV file into a Spark DataFrame.

    Parameters

    ----------

    spark : SparkSession

        The Spark session.

    logger : Logger

        Logger object for logging messages.

    file\_path : str, optional

        The path to the CSV file to load.

    Returns

    -------

    DataFrame

        DataFrame containing the data.

    Raises

    ------

    FileNotFoundError

        If the specified file does not exist.

    Notes

    -----

    This function reads data from a CSV file and loads it into a Spark DataFrame.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        # Load data from csv

        df = spark.read.option("inferSchema", "true").option(

            "header", "true").csv(file\_path)

        return df

    except *Exception* as e:

        if "Path does not exist" in *str*(e):

            logger.error(FILE\_NOT\_FOUND\_MESSAGE.format(file\_path))

            raise *FileNotFoundError*(f"File not found: {file\_path}")

        logger.error(LOAD\_DATA\_ERROR\_MESSAGE.format(*str*(e)))

        raise e

def process\_missing\_data(*df*, *logger*):

    """Process and analyze the loaded data.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the data.

    Raises

    ------

    Exception

        If an error occurs during data processing.

    Notes

    -----

    This function determines which rows in the supplied DataFrame have missing values,

    displays those rows, and removes them from the DataFrame. It details the total number of missing values,

    the cleaned DataFrame that is produced, and any problems that may have occurred.

    """

    try:

        columns\_to\_check = df.columns

        filter\_condition = None

        # Loop through the list of columns and build a filter condition to check for null values.

        for column\_name in columns\_to\_check:

            # Create a new condition to check if the column is null

            if filter\_condition is None:

                filter\_condition = col(column\_name).isNull()

            # For subsequent columns, update the filter condition to include a check for null values

            else:

                filter\_condition = filter\_condition | col(column\_name).isNull()

        missing\_data\_df = df.filter(filter\_condition)

        logger.info("Sample rows in the DataFrame with Missing Value:")

        show\_dataframe(missing\_data\_df)

        missing\_occurrence = missing\_data\_df.count()

        logger.info(

            f"There are {missing\_occurrence} rows with missing values in DataFrame.\n")

        clean\_data\_df = df.filter(~filter\_condition)

        return clean\_data\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def count\_by(*df*, *grouped\_columns*, *logger*):

    """

    Count occurrences of rows by grouping columns.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with counts, sorted in descending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function divides the data in the input DataFrame into groups according to the chosen columns,

    then counts the number of rows in each group. The outcome is a DataFrame with counts that is

    sorted in decreasing order according to the count values.

    """

    try:

        count\_by\_col = df.groupby(\*grouped\_columns).count()

        sorted\_counts\_df = count\_by\_col.orderBy("count", *ascending*=False)

        return sorted\_counts\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def percentage\_by(*df*, *grouped\_columns*, *logger*):

    """

    Calculate the percentage of occurrences by grouping columns.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with counts and percentages, sorted in descending order.

    Raises

    ------

    Exception

        If an error occurs during percentage calculation.

    Notes

    -----

    The function computes the number and percentage of occurrences within each group relative

    to the total number of rows in the DataFrame for the data in the input DataFrame, which is organized by the provided columns.

    The outcome is a DataFrame that is sorted in decreasing order using the percentage values and includes both counts and percentages.

    """

    try:

        total\_flights = df.count()

        total\_flights\_by = df.groupby(\*grouped\_columns).count()

        col\_percentage = total\_flights\_by.withColumn(

            "percentage", (total\_flights\_by["count"] / total\_flights) \* 100).orderBy("percentage", *ascending*=False)

        return col\_percentage

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def top\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *logger*):

    """

    Find the top n category.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    n : int

        Number of top category to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with the top category, sorted by count in descending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function groups data from the specified DataFrame by the specified columns and counts the number of occurrences of each category.

    It returns a DataFrame containing the top n categories with the highest counts, sorted in descending order based on count values.

    """

    try:

        top\_cat = df.groupBy(\*grouped\_columns).count().orderBy(

            "count", *ascending*=False).limit(n)

        return top\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def bottom\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *logger*):

    """

    Find the bottom n category.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    n : int

        Number of bottom category to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with the bottom category, sorted by count in ascending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function groups data from the specified DataFrame by the specified columns and counts the number of occurrences of each category.

    It returns a DataFrame containing the last n categories with the highest number, sorted in ascending order based on count values.

    """

    try:

        bottom\_cat = df.groupBy(\*grouped\_columns).count().orderBy(

            "count", *ascending*=True).limit(n)

        return bottom\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def sql\_top\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *spark*, *logger*):

    """

    Find the top n category.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    n : int

        Number of top category to retrieve.

    spark : SparkSession

        The Spark session.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with the top category, sorted by count in descending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function groups data from the specified DataFrame by the specified columns and counts the number of occurrences of each category.

    It returns a DataFrame containing the top n categories with the highest counts, sorted in descending order based on count values.

    """

    try:

        df.createOrReplaceTempView("top\_flights\_planes")

        columns\_str = ", ".join(grouped\_columns)

        query = f"""

        SELECT {columns\_str}, COUNT(\*) AS count

        FROM top\_flights\_planes

        GROUP BY {columns\_str}

        ORDER BY count DESC

        LIMIT {n}

        """

        sql\_top\_n\_cat = spark.sql(query)

        return sql\_top\_n\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def sql\_bottom\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *spark*, *logger*):

    """

    Find the bottom n category.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    n : int

        Number of bottom category to retrieve.

    spark : SparkSession

        The Spark session.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with the bottom category, sorted by count in ascending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function groups data from the specified DataFrame by the specified columns and counts the number of occurrences of each category.

    It returns a DataFrame containing the last n categories with the highest number, sorted in ascending order based on count values.

    """

    try:

        df.createOrReplaceTempView("bottom\_flights\_planes")

        columns\_str = ", ".join(grouped\_columns)

        query = f"""

        SELECT {columns\_str}, COUNT(\*) AS count

        FROM bottom\_flights\_planes

        GROUP BY {columns\_str}

        ORDER BY count ASC

        LIMIT {n}

        """

        sql\_bottom\_n\_cat = spark.sql(query)

        return sql\_bottom\_n\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def analyze\_average\_positive\_delay(*df*, *delay\_column*, *logger*):

    """

    Analyze average departure/arrival delay for positive values.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the data.

    delay\_column : str

        The column representing departure/arrival delay.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the average departure/arrival delay for positive values.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates the average departure/arrival delay for positive values in the specified column.

    It ranks the results in descending order based on the average delay.

    """

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_positive\_{suffix}\_delay"

        # Filter for positive delay values before computing the average.

        avg\_positive\_delay = df.agg(avg(when(col(delay\_column) >= 0, col(

            delay\_column))).alias(new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_positive\_delay

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

def analyze\_average\_negative\_delay(*df*, *delay\_column*, *logger*):

    """

    Analyze average departure/arrival delay for negative values.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the data.

    delay\_column : str

        The column representing departure/arrival delay.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the average departure/arrival delay for negative values.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates the average departure/arrival delay for negative values in the specified column.

    It ranks the results in descending order based on the average delay.

    """

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_negative\_{suffix}\_delay"

        # Filter for negative delay values before computing the average.

        avg\_negative\_delay = df.agg(avg(when(col(delay\_column) < 0, col(

            delay\_column))).alias(new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_negative\_delay

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

def analyze\_positive\_delay(*df*, *column*, *delay\_column*, *logger*):

    """

    Analyze positive departure/arrival delay.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the data.

    column : str

        The column by which to group the data for analysis.

    delay\_column : str

        The column representing departure/arrival delay.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the analysis of positive departure/arrival delay.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates the average positive departure/arrival delay for a specified column,

    groups the data by another column, and ranks the results in descending order based on the average positive delay.

    """

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_positive\_{suffix}\_delay"

        # Filter for positive delay values before computing the average.

        avg\_positive\_delay\_by\_column = df.groupBy(column).agg(avg(when(col(delay\_column) >= 0, col(

            delay\_column))).alias(new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_positive\_delay\_by\_column

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

def analyze\_negative\_delay(*df*, *column*, *delay\_column*, *logger*):

    """

    Analyze negative departure/arrival delay.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the data.

    column : str

        The column by which to group the data for analysis.

    delay\_column : str

        The column representing departure/arrival delay.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the analysis of negative departure/arrival delay.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates the average  departure/arrival delay for a specified column,

    groups the data by another column, and ranks the results in descending order based on the average delay.

    """

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_negative\_{suffix}\_delay"

        # Filter for negative delay values before computing the average.

        avg\_negative\_delay\_by\_column = df.groupBy(column).agg(avg(when(col(delay\_column) < 0, col(

            delay\_column))).alias(new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_negative\_delay\_by\_column

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

def numeric\_stats(*df*, *group\_by\_column*, *numeric\_column*, *logger*):

    """

    Compute statistics for a numeric column grouped by another column.

    Parameters

    ----------

    df : DataFrame

        The input DataFrame containing the data.

    group\_by\_column : str

        The name of the column to group by.

    numeric\_column : str

        The name of the numeric column to compute statistics for.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame containing statistics (average, minimum, and maximum) for the numeric column

        grouped by the specified column.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates statistics (average, minimum, maximum) for a specified numeric column in the DataFrame.

    The statistics are computed based on groups formed by the values in the specified 'group\_by\_column.'

    The resulting DataFrame is ordered in descending order of the average of the numeric column.

    """

    try:

        col\_stats = df.groupBy(group\_by\_column).agg(

            avg(numeric\_column).alias(f"average\_{numeric\_column}"),

            min(numeric\_column).alias(f"min\_{numeric\_column}"),

            max(numeric\_column).alias(f"max\_{numeric\_column}")

        ).orderBy(f"average\_{numeric\_column}", *ascending*=False)

        return col\_stats

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def compute\_flight\_speed(*df*, *distance*, *air\_time*, *logger*):

    """

    Calculate flight speed in miles per hour and add it as a new column.

    Parameters

    ----------

    df : DataFrame

        The input DataFrame containing flight data.

    distance : str

        The name of the column representing flight distance in miles.

    air\_time : str

        The name of the column representing flight air time in minutes.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame with an additional column, "flight\_speed (miles per hour)," representing the calculated

        flight speed for each record.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates the flight speed (in miles per hour) by dividing the flight distance (in miles)

    by the flight air time (in minutes) and adds it as a new column to the input DataFrame.

    Note: Flight air time is converted to hours by dividing by 60 to obtain the speed in miles per hour.

    """

    try:

        df\_add\_speed = df.withColumn(

            "flight\_speed (miles per hour)", (col(distance) / (col(air\_time) / 60)))

        return df\_add\_speed

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def shortest\_n\_flight\_from\_origin(*df*, *flight\_column*, *origin\_column*, *origin\_name*, *measurement*, *n*, *logger*):

    """

    Find the shortest 'n' flights from a specific origin based on a measurement.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the flight data.

    origin\_column : str

        The name of the column representing the flight origin.

    origin\_name : str

        The name of the origin for which to find the shortest flights.

    measurement : str

        The column name representing the measurement by which to find the shortest flights.

    n : int

        The number of shortest flights to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the 'n' shortest flights from the specified origin based on the given measurement.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function filters the DataFrame to select flights originating from a specific location (origin\_name).

    It then sorts these flights by the provided measurement column in ascending order and retrieves the top 'n' shortest flights.

    """

    try:

        origin = df.filter(df[origin\_column] == origin\_name)

        shortest\_flight = origin.select(

            flight\_column, origin\_column, measurement).orderBy(measurement, *ascending*=True).limit(n)

        return shortest\_flight

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def longest\_n\_flight\_from\_origin(*df*, *flight\_column*, *origin\_column*, *origin\_name*, *measurement*, *n*, *logger*):

    """

    Find the longest 'n' flights from a specific origin based on a measurement.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the flight data.

    origin\_column : str

        The name of the column representing the flight origin.

    origin\_name : str

        The name of the origin for which to find the longest flights.

    measurement : str

        The column name representing the measurement by which to find the longest flights.

    n : int

        The number of longest flights to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the 'n' longest flights from the specified origin based on the given measurement.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function filters the DataFrame to select flights originating from a specific location (origin\_name).

    It then sorts these flights by the provided measurement column in ascending order and retrieves the top 'n' longest flights.

    """

    try:

        origin = df.filter(df[origin\_column] == origin\_name)

        longest\_flight = origin.select(

            flight\_column, origin\_column, measurement).orderBy(measurement, *ascending*=False).limit(n)

        return longest\_flight

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def average\_duration(*df*, *carrier\_column*, *carrier\_name*, *origin\_column*, *origin\_name*, *measurement*, *logger*):

    """

    Calculate the average flight duration for a specific carrier and origin.

    Parameters

    ----------

    df : DataFrame

        The input DataFrame containing flight data.

    carrier\_column : str

        The name of the column representing the carrier.

    carrier\_name : str

        The name of the carrier for which to calculate the average duration.

    origin\_column : str

        The name of the column representing the origin airport.

    origin\_name : str

        The name of the origin airport for which to calculate the average duration.

    measurement : str

        The name of the column representing the flight duration measurement in minutes.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame with the average flight duration for the specified carrier and origin.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function filters the input DataFrame to select flights operated by a specific carrier and originating from a

    specific airport. It then calculates the average flight duration (in minutes) for these flights.

    """

    try:

        carrier = df.filter(df[carrier\_column] == carrier\_name)

        origin = carrier.filter(df[origin\_column] == origin\_name)

        average\_duration = origin.groupBy(df[carrier\_column], df[origin\_column]).agg(

            avg(measurement).alias(f"average\_{measurement} (mins)"))

        return average\_duration

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def total\_duration(*df*, *carrier\_column*, *carrier\_name*, *origin\_column*, *origin\_name*, *measurement*, *logger*):

    """

    Calculate the total flight duration for a specific carrier and origin.

    Parameters

    ----------

    df : DataFrame

        The input DataFrame containing flight data.

    carrier\_column : str

        The name of the column representing the carrier.

    carrier\_name : str

        The name of the carrier for which to calculate the total duration.

    origin\_column : str

        The name of the column representing the origin airport.

    origin\_name : str

        The name of the origin airport for which to calculate the total duration.

    measurement : str

        The name of the column representing the flight duration measurement in minutes.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame with the total flight duration for the specified carrier and origin.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function filters the input DataFrame to select flights operated by a specific carrier and originating from a

    specific airport. It then calculates the total flight duration (in hours) for these flights.

    Note: Flight air time is converted to hours by dividing by 60 to obtain the total flight duration in hours.

    """

    try:

        carrier = df.filter(df[carrier\_column] == carrier\_name)

        origin = carrier.filter(df[origin\_column] == origin\_name)

        total\_duration\_hours = origin.groupBy(df[carrier\_column], df[origin\_column]).agg(

            (sum(measurement) / 60).alias(f"total\_{measurement} (hours)"))

        return total\_duration\_hours

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def main():

    """Entry point of the script.

    Parameters

    ----------

    None

    Returns

    -------

    None

    Notes

    -----

    This function serves as the entry point of the script for processing flight data. It performs the following steps:

    1. Configures the logging settings and initializes a logger.

    2. Creates a Spark session for data processing.

    3. Loads flight data from a CSV file and processes it to handle missing values.

    4. Performs various data analyses.

    5. Displays and logs the analysis results.

    6. Stops the Spark session when processing is complete.

    """

    logger = configure\_logging()

    spark = create\_spark\_session()

    try:

        flights\_data\_frame = load\_data(spark, logger, FLIGHTS\_DATA\_FILE\_PATH)

        logger.info("Sample rows in the flights DataFrame:")

        show\_dataframe(flights\_data\_frame)

        occurrence = flights\_data\_frame.count()

        logger.info(f"There are {occurrence} occurences in the DataFrame.\n")

        logger.info(flights\_data\_frame.schema)

        clean\_flights\_data\_df = process\_missing\_data(

            flights\_data\_frame, logger)

        logger.info("Sample rows in the cleaned flights DataFrame:")

        show\_dataframe(clean\_flights\_data\_df)

        clean\_occurrence = clean\_flights\_data\_df.count()

        logger.info(

            f"{clean\_occurrence} rows remained after removing the rows with missing values.\n")

        flight\_by\_year\_month = count\_by(clean\_flights\_data\_df, [

            "year", "month"], logger)

        show\_dataframe(flight\_by\_year\_month)

        flight\_by\_day = count\_by(clean\_flights\_data\_df, ["day"], logger)

        show\_dataframe(flight\_by\_day)

        percentage\_flight\_by\_carrier = percentage\_by(

            clean\_flights\_data\_df, ["carrier"], logger)

        show\_dataframe(percentage\_flight\_by\_carrier)

        flights\_by\_origin = count\_by(clean\_flights\_data\_df, ["origin"], logger)

        show\_dataframe(flights\_by\_origin)

        flights\_by\_dest = count\_by(clean\_flights\_data\_df, ["dest"], logger)

        show\_dataframe(flights\_by\_dest)

        top\_10\_planes = top\_n\_cat\_by(

            clean\_flights\_data\_df, ["tailnum"], 10, logger)

        show\_dataframe(top\_10\_planes)

        flights\_by\_hour = count\_by(clean\_flights\_data\_df, ["hour"], logger)

        show\_dataframe(flights\_by\_hour)

        avg\_pos\_dep\_delay\_by\_carrier = analyze\_positive\_delay(

            clean\_flights\_data\_df, "carrier", "dep\_delay", logger)

        show\_dataframe(avg\_pos\_dep\_delay\_by\_carrier)

        avg\_pos\_dep\_delay = analyze\_average\_positive\_delay(

            clean\_flights\_data\_df, "dep\_delay", logger)

        show\_dataframe(avg\_pos\_dep\_delay)

        avg\_pos\_delay\_by\_month = analyze\_positive\_delay(

            clean\_flights\_data\_df, "month", "dep\_delay", logger)

        show\_dataframe(avg\_pos\_delay\_by\_month)

        avg\_pos\_delay\_by\_hour = analyze\_positive\_delay(

            clean\_flights\_data\_df, "hour", "dep\_delay", logger)

        show\_dataframe(avg\_pos\_delay\_by\_hour)

        avg\_neg\_dep\_delay\_by\_carrier = analyze\_negative\_delay(

            clean\_flights\_data\_df, "carrier", "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay\_by\_carrier)

        avg\_neg\_dep\_delay = analyze\_average\_negative\_delay(

            clean\_flights\_data\_df, "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay)

        avg\_neg\_dep\_delay\_by\_month = analyze\_negative\_delay(

            clean\_flights\_data\_df, "month", "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay\_by\_month)

        avg\_neg\_dep\_delay\_by\_hour = analyze\_negative\_delay(

            clean\_flights\_data\_df, "hour", "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay\_by\_hour)

        distance\_stats = numeric\_stats(

            clean\_flights\_data\_df, "carrier", "distance", logger)

        show\_dataframe(distance\_stats)

        transformed\_flights\_df = compute\_flight\_speed(

            clean\_flights\_data\_df, "distance", "air\_time", logger)

        show\_dataframe(transformed\_flights\_df)

        speed\_stats = numeric\_stats(

            transformed\_flights\_df, "carrier", "flight\_speed (miles per hour)", logger)

        show\_dataframe(speed\_stats)

        shortest\_flight\_distance\_PDX = shortest\_n\_flight\_from\_origin(

            transformed\_flights\_df, "flight", "origin", "PDX", "distance", 1, logger)

        show\_dataframe(shortest\_flight\_distance\_PDX)

        longest\_flight\_distance\_SEA = longest\_n\_flight\_from\_origin(

            transformed\_flights\_df, "flight", "origin", "SEA", "air\_time", 1, logger)

        show\_dataframe(longest\_flight\_distance\_SEA)

        average\_duration\_UA\_SEA = average\_duration(

            transformed\_flights\_df, "carrier", "UA", "origin", "SEA", "air\_time", logger)

        show\_dataframe(average\_duration\_UA\_SEA)

        total\_duration\_UA\_SEA = total\_duration(

            transformed\_flights\_df, "carrier", "UA", "origin", "SEA", "air\_time", logger)

        show\_dataframe(total\_duration\_UA\_SEA)

        planes\_data\_frame = load\_data(spark, logger, PLANES\_DATA\_FILE\_PATH)

        logger.info("Sample rows in the planes DataFrame:")

        show\_dataframe(planes\_data\_frame)

        occurrence = planes\_data\_frame.count()

        logger.info(f"There are {occurrence} occurences in the DataFrame.\n")

        logger.info(planes\_data\_frame.schema)

        clean\_planes\_data\_df = planes\_data\_frame.drop("speed")

        clean\_planes\_data\_df = clean\_planes\_data\_df.withColumnRenamed(

            "year", "plane\_year")

        show\_dataframe(clean\_planes\_data\_df)

        flights\_planes\_df = transformed\_flights\_df.join(

            clean\_planes\_data\_df, *on*=["tailnum"], *how*="inner")

        show\_dataframe(flights\_planes\_df)

        occurrence = flights\_planes\_df.count()

        logger.info(f"There are {occurrence} occurences in the DataFrame.\n")

        clean\_flights\_planes\_data\_df = process\_missing\_data(

            flights\_planes\_df, logger)

        logger.info("Sample rows in the cleaned planes DataFrame:")

        show\_dataframe(clean\_flights\_planes\_data\_df)

        clean\_occurrence = clean\_flights\_planes\_data\_df.count()

        logger.info(

            f"{clean\_occurrence} rows remained after removing the rows with missing values.\n")

        top\_20\_flights\_planes = top\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, logger)

        show\_dataframe(top\_20\_flights\_planes)

        bottom\_20\_flights\_planes = bottom\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, logger)

        show\_dataframe(bottom\_20\_flights\_planes)

        sql\_top\_20\_flights\_planes = sql\_top\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, spark, logger)

        show\_dataframe(sql\_top\_20\_flights\_planes)

        sql\_bottom\_20\_flights\_planes = sql\_bottom\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, spark, logger)

        show\_dataframe(sql\_bottom\_20\_flights\_planes)

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

    finally:

        if spark is not None:

            spark.stop()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**(a)**

***Script Constants Snippet***

# Constants

SCRIPTS\_DIR = os.path.abspath(os.path.dirname(\_\_file\_\_))

DATA\_DIR = os.path.join(SCRIPTS\_DIR, "..", "data")

FLIGHTS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "flights\_data\_v2.csv")

PLANES\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "planes\_data\_v2.csv")

LOGGING\_LEVEL = logging.INFO

LOAD\_DATA\_ERROR\_MESSAGE = "An error occurred while loading data: {}"

FILE\_NOT\_FOUND\_MESSAGE = "The specified file does not exist: {}"

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def load\_data(*spark*, *logger*, *file\_path*):

    try:

        # Load data from csv

        df = spark.read.option("inferSchema", "true").option(

            "header", "true").csv(file\_path)

        return df

    except *Exception* as e:

        if "Path does not exist" in *str*(e):

            logger.error(FILE\_NOT\_FOUND\_MESSAGE.format(file\_path))

            raise *FileNotFoundError*(f"File not found: {file\_path}")

        logger.error(LOAD\_DATA\_ERROR\_MESSAGE.format(*str*(e)))

        raise e

def process\_missing\_data(*df*, *logger*):

    try:

        columns\_to\_check = df.columns

        filter\_condition = None

        # Loop through the list of columns and build a filter condition to check for null values.

        for column\_name in columns\_to\_check:

            # Create a new condition to check if the column is null

            if filter\_condition is None:

                filter\_condition = col(column\_name).isNull()

            # For subsequent columns, update the filter condition to include a check for null values

            else:

                filter\_condition = filter\_condition | col(column\_name).isNull()

        missing\_data\_df = df.filter(filter\_condition)

        logger.info("Sample rows in the DataFrame with Missing Value:")

        show\_dataframe(missing\_data\_df)

        missing\_occurrence = missing\_data\_df.count()

        logger.info(

            f"There are {missing\_occurrence} rows with missing values in DataFrame.\n")

        clean\_data\_df = df.filter(~filter\_condition)

        return clean\_data\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        flights\_data\_frame = load\_data(spark, logger, FLIGHTS\_DATA\_FILE\_PATH)

        logger.info("Sample rows in the flights DataFrame:")

        show\_dataframe(flights\_data\_frame)

        occurrence = flights\_data\_frame.count()

        logger.info(f"There are {occurrence} occurences in the DataFrame.\n")

        logger.info(flights\_data\_frame.schema)

        clean\_flights\_data\_df = process\_missing\_data(

            flights\_data\_frame, logger)

        logger.info("Sample rows in the cleaned flights DataFrame:")

        show\_dataframe(clean\_flights\_data\_df)

        clean\_occurrence = clean\_flights\_data\_df.count()

        logger.info(

            f"{clean\_occurrence} rows remained after removing the rows with missing values.\n")

***Output Screenshots***

A screen shot of a computer screen

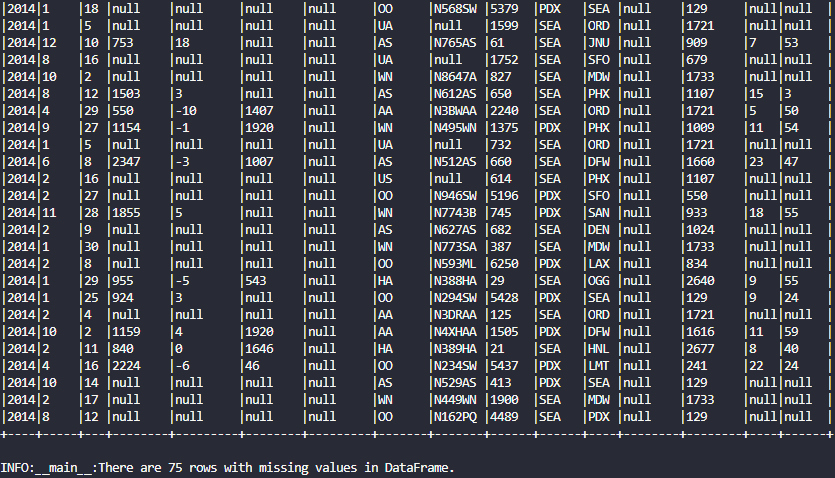
Description automatically generated

*Figure 12. Content, number of occurrences and schema of flights DataFrame.*

**A screen shot of a computer screen

Description automatically generated**

*Figure 13. Content and number of occurrences of missing data part 1.*

****

*Figure 14. Content and number of occurrences of missing data part 2.*

**A screenshot of a computer screen

Description automatically generated**

*Figure 15. Content and number of occurrences of the Cleaned flights DataFrame.*

**(b)**

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def count\_by(*df*, *grouped\_columns*, *logger*):

    try:

        count\_by\_col = df.groupby(\*grouped\_columns).count()

        sorted\_counts\_df = count\_by\_col.orderBy("count", *ascending*=False)

        return sorted\_counts\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def percentage\_by(*df*, *grouped\_columns*, *logger*):

        total\_flights = df.count()

        total\_flights\_by = df.groupby(\*grouped\_columns).count()

        col\_percentage = total\_flights\_by.withColumn(

            "percentage", (total\_flights\_by["count"] / total\_flights) \* 100).orderBy("percentage", *ascending*=False)

        return col\_percentage

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def top\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *logger*):

    try:

        top\_cat = df.groupBy(\*grouped\_columns).count().orderBy(

            "count", *ascending*=False).limit(n)

        return top\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        flight\_by\_year\_month = count\_by(clean\_flights\_data\_df, [

            "year", "month"], logger)

        show\_dataframe(flight\_by\_year\_month)

        flight\_by\_day = count\_by(clean\_flights\_data\_df, ["day"], logger)

        show\_dataframe(flight\_by\_day)

        percentage\_flight\_by\_carrier = percentage\_by(

            clean\_flights\_data\_df, ["carrier"], logger)

        show\_dataframe(percentage\_flight\_by\_carrier)

        flights\_by\_origin = count\_by(clean\_flights\_data\_df, ["origin"], logger)

        show\_dataframe(flights\_by\_origin)

        flights\_by\_dest = count\_by(clean\_flights\_data\_df, ["dest"], logger)

        show\_dataframe(flights\_by\_dest)

        top\_10\_planes = top\_n\_cat\_by(

            clean\_flights\_data\_df, ["tailnum"], 10, logger)

        show\_dataframe(top\_10\_planes)

***Output Screenshots***

**A screenshot of a computer screen

Description automatically generated**

*Figure 16. Sorted number of flights per year and month.*

**A screenshot of a computer screen

Description automatically generated**

*Figure 17. Sorted number of flights per day.*

**A screenshot of a computer screen

Description automatically generated**

*Figure 18. Sorted number and percentage of flights per carrier.*

**A screenshot of a computer code

Description automatically generated**

*Figure 19. Sorted number of flights per origin airport.*

**A screenshot of a computer screen

Description automatically generated** **A screen shot of a computer screen

Description automatically generated**

*Figure 20. Sorted number of flights per destination.*

**A screen shot of a computer

Description automatically generated**

*Figure 21. Sorted top 10 of flights per plane.*

**(c)**

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def count\_by(*df*, *grouped\_columns*, *logger*):

    try:

        count\_by\_col = df.groupby(\*grouped\_columns).count()

        sorted\_counts\_df = count\_by\_col.orderBy("count", *ascending*=False)

        return sorted\_counts\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def analyze\_average\_positive\_delay(*df*, *delay\_column*, *logger*):

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_positive\_{suffix}\_delay"

        # Filter for positive delay values before computing the average.

        avg\_positive\_delay = df.agg(avg(when(col(delay\_column) >= 0, col(

            delay\_column))).alias(new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_positive\_delay

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

def analyze\_average\_negative\_delay(*df*, *delay\_column*, *logger*):

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_negative\_{suffix}\_delay"

        # Filter for negative delay values before computing the average.

        avg\_negative\_delay = df.agg(avg(when(col(delay\_column) < 0, col(

            delay\_column))).alias(new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_negative\_delay

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

def analyze\_positive\_delay(*df*, *column*, *delay\_column*, *logger*):

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_positive\_{suffix}\_delay"

        # Filter for positive delay values before computing the average.

        avg\_positive\_delay\_by\_column = df.groupBy(column).agg(avg(when(col(delay\_column) >= 0, col(

            delay\_column))).alias(new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_positive\_delay\_by\_column

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

def analyze\_negative\_delay(*df*, *column*, *delay\_column*, *logger*):

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_negative\_{suffix}\_delay"

        # Filter for negative delay values before computing the average.

        avg\_negative\_delay\_by\_column = df.groupBy(column).agg(avg(when(col(delay\_column) < 0, col(

            delay\_column))).alias(new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_negative\_delay\_by\_column

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        flights\_by\_hour = count\_by(clean\_flights\_data\_df, ["hour"], logger)

        show\_dataframe(flights\_by\_hour)

        avg\_pos\_dep\_delay\_by\_carrier = analyze\_positive\_delay(

            clean\_flights\_data\_df, "carrier", "dep\_delay", logger)

        show\_dataframe(avg\_pos\_dep\_delay\_by\_carrier)

        avg\_pos\_dep\_delay = analyze\_average\_positive\_delay(

            clean\_flights\_data\_df, "dep\_delay", logger)

        show\_dataframe(avg\_pos\_dep\_delay)

        avg\_pos\_delay\_by\_month = analyze\_positive\_delay(

            clean\_flights\_data\_df, "month", "dep\_delay", logger)

        show\_dataframe(avg\_pos\_delay\_by\_month)

        avg\_pos\_delay\_by\_hour = analyze\_positive\_delay(

            clean\_flights\_data\_df, "hour", "dep\_delay", logger)

        show\_dataframe(avg\_pos\_delay\_by\_hour)

        avg\_neg\_dep\_delay\_by\_carrier = analyze\_negative\_delay(

            clean\_flights\_data\_df, "carrier", "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay\_by\_carrier)

        avg\_neg\_dep\_delay = analyze\_average\_negative\_delay(

            clean\_flights\_data\_df, "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay)

        avg\_neg\_dep\_delay\_by\_month = analyze\_negative\_delay(

            clean\_flights\_data\_df, "month", "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay\_by\_month)

        avg\_neg\_dep\_delay\_by\_hour = analyze\_negative\_delay(

            clean\_flights\_data\_df, "hour", "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay\_by\_hour)

***Output Screenshots***

**A screenshot of a computer screen

Description automatically generated**

*Figure 22. Sorted number of flights per hour.*

**A screen shot of a computer

Description automatically generated**

*Figure 23. Sorted average positive departure delay per carrier.*

**A screenshot of a computer screen

Description automatically generated**

*Figure 24. Average positive departure delay.*

**A screenshot of a computer program

Description automatically generated**

*Figure 25. Sorted average positive departure delay per month.*

**A screenshot of a computer screen

Description automatically generated**

*Figure 26. Sorted average positive departure delay per hour.*

**A computer screen shot of a number

Description automatically generated**

*Figure 27. Sorted average negative departure delay per carrier.*

**A screenshot of a computer code

Description automatically generated**

*Figure 28. Average negative departure delay.*

*A screenshot of a computer program

Description automatically generated*

*Figure 29. Sorted average negative departure delay per month.*

*A screenshot of a computer program

Description automatically generated*

*Figure 30. Sorted average negative departure delay per hour.*

**(d)**

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def numeric\_stats(*df*, *group\_by\_column*, *numeric\_column*, *logger*):

    try:

        col\_stats = df.groupBy(group\_by\_column).agg(

            avg(numeric\_column).alias(f"average\_{numeric\_column}"),

            min(numeric\_column).alias(f"min\_{numeric\_column}"),

            max(numeric\_column).alias(f"max\_{numeric\_column}")

        ).orderBy(f"average\_{numeric\_column}", *ascending*=False)

        return col\_stats

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def compute\_flight\_speed(*df*, *distance*, *air\_time*, *logger*):

    try:

        df\_add\_speed = df.withColumn(

            "flight\_speed (miles per hour)", (col(distance) / (col(air\_time) / 60)))

        return df\_add\_speed

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def shortest\_n\_flight\_from\_origin(*df*, *flight\_column*, *origin\_column*, *origin\_name*, *measurement*, *n*, *logger*):

    try:

        origin = df.filter(df[origin\_column] == origin\_name)

        shortest\_flight = origin.select(

            flight\_column, origin\_column, measurement).orderBy(measurement, *ascending*=True).limit(n)

        return shortest\_flight

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def longest\_n\_flight\_from\_origin(*df*, *flight\_column*, *origin\_column*, *origin\_name*, *measurement*, *n*, *logger*):

    try:

        origin = df.filter(df[origin\_column] == origin\_name)

        longest\_flight = origin.select(

            flight\_column, origin\_column, measurement).orderBy(measurement, *ascending*=False).limit(n)

        return longest\_flight

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def average\_duration(*df*, *carrier\_column*, *carrier\_name*, *origin\_column*, *origin\_name*, *measurement*, *logger*):

    try:

        carrier = df.filter(df[carrier\_column] == carrier\_name)

        origin = carrier.filter(df[origin\_column] == origin\_name)

        average\_duration = origin.groupBy(df[carrier\_column], df[origin\_column]).agg(

            avg(measurement).alias(f"average\_{measurement} (mins)"))

        return average\_duration

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def total\_duration(*df*, *carrier\_column*, *carrier\_name*, *origin\_column*, *origin\_name*, *measurement*, *logger*):

    try:

        carrier = df.filter(df[carrier\_column] == carrier\_name)

        origin = carrier.filter(df[origin\_column] == origin\_name)

        total\_duration\_hours = origin.groupBy(df[carrier\_column], df[origin\_column]).agg(

            (sum(measurement) / 60).alias(f"total\_{measurement} (hours)"))

        return total\_duration\_hours

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        distance\_stats = numeric\_stats(

            clean\_flights\_data\_df, "carrier", "distance", logger)

        show\_dataframe(distance\_stats)

        transformed\_flights\_df = compute\_flight\_speed(

            clean\_flights\_data\_df, "distance", "air\_time", logger)

        show\_dataframe(transformed\_flights\_df)

        speed\_stats = numeric\_stats(

            transformed\_flights\_df, "carrier", "flight\_speed (miles per hour)", logger)

        show\_dataframe(speed\_stats)

        shortest\_flight\_distance\_PDX = shortest\_n\_flight\_from\_origin(

            transformed\_flights\_df, "flight", "origin", "PDX", "distance", 1, logger)

        show\_dataframe(shortest\_flight\_distance\_PDX)

        longest\_flight\_distance\_SEA = longest\_n\_flight\_from\_origin(

            transformed\_flights\_df, "flight", "origin", "SEA", "air\_time", 1, logger)

        show\_dataframe(longest\_flight\_distance\_SEA)

        average\_duration\_UA\_SEA = average\_duration(

            transformed\_flights\_df, "carrier", "UA", "origin", "SEA", "air\_time", logger)

        show\_dataframe(average\_duration\_UA\_SEA)

        total\_duration\_UA\_SEA = total\_duration(

            transformed\_flights\_df, "carrier", "UA", "origin", "SEA", "air\_time", logger)

        show\_dataframe(total\_duration\_UA\_SEA)

***Output Screenshots***

**A screen shot of a computer

Description automatically generated**

*Figure 31. Average, minimum, and maximum flight distance by carrier, sorted by average distance.*

**A screen shot of a computer screen

Description automatically generated**

*Figure 32. New flight speed (miles per hour) calculated column.*

**A computer screen shot of a number

Description automatically generated**

*Figure 33. Average, minimum, and maximum flight speed (miles per hour) by carrier, sorted by average speed.*

A screen shot of a computer screen

Description automatically generated

*Figure 34. Shortest distance flight from PDX.*

**A screen shot of a computer

Description automatically generated**

*Figure 35. Longest duration flight from SEA.*

**A screen shot of a computer code

Description automatically generated**

*Figure 36. Average flight duration of carrier “UA” originiated from “SEA”.*

**A screen shot of a computer code

Description automatically generated**

*Figure 37. Total flight duration of carrier “UA” originiated from “SEA”.*

**(e)**

***Script Constants Snippet***

# Constants

SCRIPTS\_DIR = os.path.abspath(os.path.dirname(\_\_file\_\_))

DATA\_DIR = os.path.join(SCRIPTS\_DIR, "..", "data")

FLIGHTS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "flights\_data\_v2.csv")

PLANES\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "planes\_data\_v2.csv")

LOGGING\_LEVEL = logging.INFO

LOAD\_DATA\_ERROR\_MESSAGE = "An error occurred while loading data: {}"

FILE\_NOT\_FOUND\_MESSAGE = "The specified file does not exist: {}"

***Script Functions Snippet***

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def load\_data(*spark*, *logger*, *file\_path*):

    try:

        # Load data from csv

        df = spark.read.option("inferSchema", "true").option(

            "header", "true").csv(file\_path)

        return df

    except *Exception* as e:

        if "Path does not exist" in *str*(e):

            logger.error(FILE\_NOT\_FOUND\_MESSAGE.format(file\_path))

            raise *FileNotFoundError*(f"File not found: {file\_path}")

        logger.error(LOAD\_DATA\_ERROR\_MESSAGE.format(*str*(e)))

        raise e

def process\_missing\_data(*df*, *logger*):

    try:

        columns\_to\_check = df.columns

        filter\_condition = None

        # Loop through the list of columns and build a filter condition to check for null values.

        for column\_name in columns\_to\_check:

            # Create a new condition to check if the column is null

            if filter\_condition is None:

                filter\_condition = col(column\_name).isNull()

            # For subsequent columns, update the filter condition to include a check for null values

            else:

                filter\_condition = filter\_condition | col(column\_name).isNull()

        missing\_data\_df = df.filter(filter\_condition)

        logger.info("Sample rows in the DataFrame with Missing Value:")

        show\_dataframe(missing\_data\_df)

        missing\_occurrence = missing\_data\_df.count()

        logger.info(

            f"There are {missing\_occurrence} rows with missing values in DataFrame.\n")

        clean\_data\_df = df.filter(~filter\_condition)

        return clean\_data\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def top\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *logger*):

    try:

        top\_cat = df.groupBy(\*grouped\_columns).count().orderBy(

            "count", *ascending*=False).limit(n)

        return top\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def bottom\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *logger*):

    try:

        bottom\_cat = df.groupBy(\*grouped\_columns).count().orderBy(

            "count", *ascending*=True).limit(n)

        return bottom\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def sql\_top\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *spark*, *logger*):

    try:

        df.createOrReplaceTempView("top\_flights\_planes")

        columns\_str = ", ".join(grouped\_columns)

        query = f"""

        SELECT {columns\_str}, COUNT(\*) AS count

        FROM top\_flights\_planes

        GROUP BY {columns\_str}

        ORDER BY count DESC

        LIMIT {n}

        """

        sql\_top\_n\_cat = spark.sql(query)

        return sql\_top\_n\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def sql\_bottom\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *spark*, *logger*):

    try:

        df.createOrReplaceTempView("bottom\_flights\_planes")

        columns\_str = ", ".join(grouped\_columns)

        query = f"""

        SELECT {columns\_str}, COUNT(\*) AS count

        FROM bottom\_flights\_planes

        GROUP BY {columns\_str}

        ORDER BY count ASC

        LIMIT {n}

        """

        sql\_bottom\_n\_cat = spark.sql(query)

        return sql\_bottom\_n\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        planes\_data\_frame = load\_data(spark, logger, PLANES\_DATA\_FILE\_PATH)

        logger.info("Sample rows in the planes DataFrame:")

        show\_dataframe(planes\_data\_frame)

        occurrence = planes\_data\_frame.count()

        logger.info(f"There are {occurrence} occurences in the DataFrame.\n")

        logger.info(planes\_data\_frame.schema)

        clean\_planes\_data\_df = planes\_data\_frame.drop("speed")

        clean\_planes\_data\_df = clean\_planes\_data\_df.withColumnRenamed(

            "year", "plane\_year")

        show\_dataframe(clean\_planes\_data\_df)

        flights\_planes\_df = transformed\_flights\_df.join(

            clean\_planes\_data\_df, *on*=["tailnum"], *how*="inner")

        show\_dataframe(flights\_planes\_df)

        occurrence = flights\_planes\_df.count()

        logger.info(f"There are {occurrence} occurences in the DataFrame.\n")

        clean\_flights\_planes\_data\_df = process\_missing\_data(

            flights\_planes\_df, logger)

        logger.info("Sample rows in the cleaned planes DataFrame:")

        show\_dataframe(clean\_flights\_planes\_data\_df)

        clean\_occurrence = clean\_flights\_planes\_data\_df.count()

        logger.info(

            f"{clean\_occurrence} rows remained after removing the rows with missing values.\n")

        top\_20\_flights\_planes = top\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, logger)

        show\_dataframe(top\_20\_flights\_planes)

        bottom\_20\_flights\_planes = bottom\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, logger)

        show\_dataframe(bottom\_20\_flights\_planes)

        sql\_top\_20\_flights\_planes = sql\_top\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, spark, logger)

        show\_dataframe(sql\_top\_20\_flights\_planes)

        sql\_bottom\_20\_flights\_planes = sql\_bottom\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, spark, logger)

        show\_dataframe(sql\_bottom\_20\_flights\_planes)

***Output Screenshots***

***A black background with white dots

Description automatically generated***

*Figure 38. Content, number of occurrences and schema of planes DataFrame.*

**A screenshot of a computer program

Description automatically generated**

*Figure 39. Planes DataFrame without speed column and renamed year column header.*

**A screen shot of a computer

Description automatically generated**

*Figure 40. Content and number of occurrences of joined dataset.*

**A screen shot of a computer screen

Description automatically generated**

*Figure 41. Content and number of occurrences of missing data part 1.*

**A close up of a screen

Description automatically generated**

*Figure 42. Content and number of occurrences of missing data part 2.*

*A screen shot of a computer screen

Description automatically generated*

*Figure 43. Content and number of occurrences of the Cleaned planes DataFrame.*

**A screenshot of a computer code

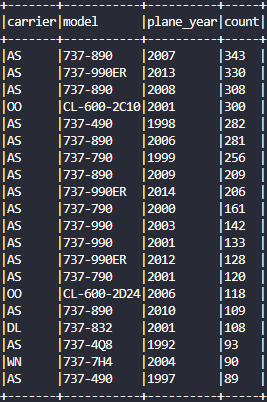
Description automatically generated**

*Figure 44. Sorted top 20 carrier, model, and plane year with the most number of trips.*

**A screenshot of a computer code

Description automatically generated**

*Figure 45. Sorted bottom 20 carrier, model, and plane year with the least number of trips.*

****

*Figure 46. Sorted top 20 carrier, model, and plane year with the most number of trips (SQL approach).*

**A screenshot of a computer code

Description automatically generated**

*Figure 45. Sorted bottom 20 carrier, model, and plane year with the least number of trips (SQL approach).*

**Question 4**

**Full PySpark Program**

import logging

import os

from pyspark import SparkContext

# Constants

SCRIPTS\_DIR = os.path.abspath(os.path.dirname(\_\_file\_\_))

DATA\_DIR = os.path.join(SCRIPTS\_DIR, "..", "data")

GROCERY\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "grocery\_data.csv")

LOGGING\_LEVEL = logging.INFO

LOAD\_DATA\_ERROR\_MESSAGE = "An error occurred while loading data: {}"

FILE\_NOT\_FOUND\_MESSAGE = "The specified file does not exist: {}"

def configure\_logging():

    """Configure logging settings and return a logger object.

    Returns

    -------

    logger : object

        Logger object for logging messages.

    Notes

    -----

    This function returns a logger object that may be used to log messages

    inside the program and initializes the logging parameters, including the logging level.

    The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    logging.basicConfig(*level*=LOGGING\_LEVEL)

    return logging.getLogger(\_\_name\_\_)

def create\_spark\_context(*app\_name*="TMA\_Market\_Basket\_Analysis"):

    """

    Create and return a SparkContext.

    Parameters

    ----------

    app\_name : str, optional

        The name of the Spark application. Default is "TMA\_Market\_Basket\_Analysis".

    Returns

    -------

    SparkContext

        An instance of SparkContext.

    Notes

    -----

    This function initializes a SparkContext, which is the entry point for Spark operations.

    """

    sc = SparkContext("local", app\_name)

    return sc

def show\_rdd(*rdd*, *logger*, *max\_rows*=100, *show\_rows*=20):

    """

    Show rows of a rdd with the option to limit the number of rows displayed.

    Parameters

    ----------

    df : rdd

        The rdd to be displayed.

    max\_rows : int, optional

        The maximum number of rows to display. Default is 100.

    show\_rows : int, optional

        The rows to display if records is above max rows. Default is 20.

    Returns

    -------

    None

    Notes

    -----

    This function displays rows of the input rdd. If the rdd contains more

    rows than the specified `max\_rows`, it will limit the display to the first `show\_rows`

    rows. If the rdd has fewer rows than `max\_rows`, it will display all available

    rows without truncation.

    """

    if rdd.count() > max\_rows:

        logger.info(rdd.take(show\_rows))

    else:

        logger.info(rdd.collect())

def load\_data(*sc*, *logger*, *file\_path*):

    """

    Load data from CSV file into a Spark RDD.

    Parameters

    ----------

    spark : SparkContext

        The Spark context.

    logger : object

        Logger object for logging messages.

    file\_path : str, optional

        The path to the CSV file to load.

    Returns

    -------

    RDD

        RDD containing the data.

    Raises

    ------

    FileNotFoundError

        If the specified file does not exist.

    Notes

    -----

    This function reads data from a CSV file and loads it into a Spark RDD.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        rdd = sc.textFile(file\_path)

        return rdd

    except *Exception* as e:

        if "Path does not exist" in *str*(e):

            logger.error(FILE\_NOT\_FOUND\_MESSAGE.format(file\_path))

            raise *FileNotFoundError*(f"File not found: {file\_path}")

        logger.error(LOAD\_DATA\_ERROR\_MESSAGE.format(*str*(e)))

        raise e

def cleanse(*rdd*, *logger*):

    """

    Cleanse data in an RDD by stripping whitespace and converting to lowercase.

    Parameters

    ----------

    rdd : RDD

        The input RDD containing data to be cleansed.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        RDD containing the cleansed data.

    Raises

    ------

    Exception

        If an error occurs during the cleansing process, an exception is raised.

    Notes

    -----

    This function takes an RDD as input and performs the following cleansing operations on each row:

    1. Remove leading and trailing whitespace.

    2. Convert all text to lowercase.

    The cleansed data is returned as an RDD.

    """

    try:

        cleansed\_rdd = rdd.map(

            lambda *x*: [item.strip().lower() for item in x.split(',')])

        return cleansed\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def transaction\_with\_most\_items(*cleanse\_rdd*, *logger*):

    """

    Find the transaction with the most items in an RDD.

    Parameters

    ----------

    cleanse\_rdd : RDD

        The RDD containing cleansed transaction data.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    tuple

        A tuple containing the content of the transaction with the most items and the number of items in that transaction.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function takes an RDD of cleansed transaction data and finds the transaction with the highest number of items.

    It returns a tuple containing the content of the transaction and the count of items in that transaction.

    """

    try:

        transaction\_item\_count = cleanse\_rdd.map(

            lambda *items*: (items, len(items)))

        max\_transaction = transaction\_item\_count.max(*key*=lambda *x*: x[1])

        max\_transaction\_content, max\_transaction\_items = max\_transaction

        return max\_transaction\_content, max\_transaction\_items

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def count\_unique\_items(*cleanse\_rdd*, *logger*):

    """

    Count the number of unique items in an RDD of transaction data.

    Parameters

    ----------

    cleanse\_rdd : RDD

        The RDD containing cleansed transaction data.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    int

        The count of unique items in the RDD.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function takes an RDD of cleansed transaction data, flattens it to extract individual items, and then counts the number

    of unique items in the RDD.

    """

    try:

        unique\_items = cleanse\_rdd.flatMap(lambda *items*: items).distinct()

        count = unique\_items.count()

        return count

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def top\_n\_item\_with\_percentage(*cleanse\_rdd*, *n*, *logger*):

    """

    Calculate the top N most frequent items along with their occurrence percentages.

    Parameters

    ----------

    cleanse\_rdd : RDD

        The RDD containing cleansed transaction data.

    n : int

        The number of top items to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    list

        A list of tuples containing the top N items, their occurrence counts, and occurrence percentages.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function takes an RDD of cleansed transaction data, calculates the most frequent N items, and computes their

    occurrence percentages relative to the total number of transactions.

    """

    try:

        item = cleanse\_rdd.flatMap(lambda *items*: items)

        item\_count = item.countByValue().items()

        sorted\_item\_count = sorted(

            item\_count, *key*=(lambda *x*: x[1]), *reverse*=True)

        top\_item = sorted\_item\_count[:n]

        total\_transaction = item.count()

        top\_items\_with\_percentage = [

            (item, count, ((count / total\_transaction) \* 100)) for item, count in top\_item]

        return top\_items\_with\_percentage

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def bottom\_n\_item\_with\_percentage(*cleanse\_rdd*, *n*, *logger*):

    """

    Calculate the bottom N least frequent items along with their occurrence percentages.

    Parameters

    ----------

    cleanse\_rdd : RDD

        The RDD containing cleansed transaction data.

    n : int

        The number of bottom items to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    list

        A list of tuples containing the bottom N items, their occurrence counts, and occurrence percentages.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function takes an RDD of cleansed transaction data, calculates the least frequent N items, and computes their

    occurrence percentages relative to the total number of transactions.

    """

    try:

        item = cleanse\_rdd.flatMap(lambda *items*: items)

        item\_count = item.countByValue().items()

        sorted\_item\_count = sorted(

            item\_count, *key*=(lambda *x*: x[1]), *reverse*=False)

        bottom\_item = sorted\_item\_count[:n]

        total\_transaction = item.count()

        bottom\_items\_with\_percentage = [

            (item, count, ((count / total\_transaction) \* 100)) for item, count in bottom\_item]

        return bottom\_items\_with\_percentage

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def add\_index(*cleanse\_rdd*, *logger*):

    """

    Add an index to each transaction of the RDD.

    Parameters

    ----------

    cleanse\_rdd : RDD

        The RDD containing cleansed data.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        RDD containing the data with added indices.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function takes an RDD and adds a unique index to each transaction. The resulting RDD contains the original data

    along with the corresponding index.

    """

    try:

        indexed\_rdd = cleanse\_rdd.zipWithIndex()

        return indexed\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def generate\_combinations(*indexed\_rdd*, *logger*):

    """

    Generate combinations of two grocery items within each transaction.

    Parameters

    ----------

    indexed\_rdd : RDD

        The RDD containing data with added indices.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        RDD containing combinations of two items and their corresponding transaction indices.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function takes an RDD with added indices and generates all possible combinations of two grocery items within each transaction.

    The resulting RDD contains the item pairs and their corresponding transaction indices.

    """

    try:

        item\_combinations\_rdd = indexed\_rdd.flatMap(lambda *transaction*: [(

            (item1, item2), transaction[1])

            for item1 in transaction[0] for item2 in transaction[0]

            # Ensure item1 is less than item2 (in alphabetical order) to avoid duplicate pairs

            if item1 < item2

        ])

        sorted\_item\_combinations\_rdd = item\_combinations\_rdd.sortBy(

            lambda *x*: (x[1], x[0]))

        return sorted\_item\_combinations\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def association(*combinations\_rdd*, *logger*):

    """

    Generate associations between grocery items and their transaction indices.

    Parameters

    ----------

    combinations\_rdd : RDD

        RDD containing combinations of grocery items and their corresponding transaction indices.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        RDD containing associations between item combinations and their transaction indices.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function takes an RDD with item combinations and their transaction indices and generates associations.

    The resulting RDD contains the associations between grocery item combinations and their transaction indices.

    """

    try:

        transaction\_indices = combinations\_rdd.groupByKey().map(

            lambda *x*: (x[0], *list*(x[-1])))

        return transaction\_indices

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def item\_pair\_counts(*association\_rdd*, *logger*):

    """

    Calculate the counts of item pairs in association data.

    Parameters

    ----------

    association\_rdd : RDD

        RDD containing associations between item combinations and their transaction indices.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        RDD containing the counts of item pairs sorted by count in descending order.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function takes an RDD with item associations and calculates the counts of item pairs.

    The resulting RDD contains the counts of item pairs, sorted in descending order by count.

    """

    try:

        item\_count\_rdd = association\_rdd.map(lambda *x*: (x[0], len(x[1])))

        sorted\_item\_count\_rdd = item\_count\_rdd.sortBy(

            lambda *x*: x[1], *ascending*=False)

        return sorted\_item\_count\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def support(*item\_pair\_counts\_rdd*, *logger*):

    """

    Calculate the support of item pairs in association data.

    Parameters

    ----------

    item\_pair\_counts\_rdd : RDD

        RDD containing the counts of item pairs sorted by count.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        RDD containing the support of item pairs.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function calculates the support of item pairs based on their counts.

    """

    try:

        total\_records = item\_pair\_counts\_rdd.map(lambda *x*: x[1]).sum()

        support\_rdd = item\_pair\_counts\_rdd.map(lambda *x*: (

            (x[0], (x[1], x[1] / total\_records))))

        return support\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def top\_n\_item\_pair\_with\_support(*support\_rdd*, *n*, *logger*):

    """

    Get the top N item pairs with the highest occurrence count.

    Parameters

    ----------

    support\_rdd : RDD

        RDD containing item pairs and their support as a percentage of total records.

    n : int

        The number of top item pairs to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    list

        A list of the top N item pairs with the occurrence count and support value, sorted by occurence count.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function retrieves the top N item pairs with the highest occurrence count.

    """

    try:

        sorted\_items = support\_rdd.sortBy(

            lambda *x*: x[1][0], *ascending*=False).take(n)

        return sorted\_items

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def bottom\_n\_item\_pair\_with\_support(*support\_rdd*, *n*, *logger*):

    """

    Get the bottom N item pairs with the lowest occurrence count.

    Parameters

    ----------

    support\_rdd : RDD

        RDD containing item pairs and their support as a percentage of total records.

    n : int

        The number of bottom item pairs to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    list

        A list of the bottom N item pairs with the occurrence count and support value, sorted by occurence count.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function retrieves the bottom N item pairs with the lowest occurrence count.

    """

    try:

        sorted\_items = support\_rdd.sortBy(

            lambda *x*: x[1][0], *ascending*=True).take(n)

        return sorted\_items

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def generate\_permutations(*indexed\_rdd*, *logger*):

    """

    Generate all permutations of item pairs within each transaction.

    Parameters

    ----------

    indexed\_rdd : RDD

        RDD containing indexed transactions, each as a list of items and a transaction index.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        RDD containing sorted item permutations within each transaction.

    Raises

    ------

    Exception

        If an error occurs during the process, an exception is raised.

    Notes

    -----

    This function generates all possible permutations of item pairs within each transaction in the RDD.

    It takes an RDD of indexed transactions, where each transaction is represented as a list of items and a unique transaction index.

    For each transaction, it computes permutations of item pairs, ensuring that item pairs are sorted lexicographically by their names.

    The output RDD contains these sorted item permutations for each transaction.

    """

    try:

        def generate\_pairs(*transaction*):

            item\_list, transaction\_index = transaction

            item\_pairs = []

            # Iterate through items in the transaction

            for i in range(len(item\_list)):

                for j in range(i + 1, len(item\_list)):

                    item1 = item\_list[i]

                    item2 = item\_list[j]

                    # Create pairs for item1 and item2, as well as their reverse order

                    item\_pairs.append(((item1, item2), transaction\_index))

                    item\_pairs.append(((item2, item1), transaction\_index))

            return item\_pairs

        permutations\_rdd = indexed\_rdd.flatMap(generate\_pairs)

        sorted\_permutations\_rdd = permutations\_rdd.sortBy(

            lambda *x*: (x[1], x[0]))

        return sorted\_permutations\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def item\_count(*cleansed\_rdd*, *logger*):

    """

    Count the occurrences of each item in the RDD.

    Parameters

    ----------

    cleansed\_rdd : RDD

        RDD containing cleansed transaction data.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    dict

        A dictionary where keys are items, and values are their counts.

    Raises

    ------

    Exception

        If any error occurs during the counting process.

    Notes

    -----

    This function counts the occurrences of each unique item in the provided RDD.

    The result is returned as a dictionary where item names are keys and their counts are values.

    """

    try:

        # Transform each transaction into a list of (item, 1) pairs, then reduce to count occurrences.

        item\_frequencies\_rdd = cleansed\_rdd.flatMap(lambda *transaction*: [(

            item, 1) for item in transaction]).reduceByKey(lambda *a*, *b*: a + b)

        sorted\_item\_counts\_rdd = item\_frequencies\_rdd.sortBy(

            lambda *x*: x[1], *ascending*=False)

        sorted\_item\_counts\_dict = *dict*(sorted\_item\_counts\_rdd.collect())

        return sorted\_item\_counts\_dict

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def confidence(*frequency\_xy\_rdd*, *frequency\_x\_dict*, *logger*):

    """

    Calculate the confidence of item pairs based on their frequencies.

    Parameters

    ----------

    frequency\_xy\_rdd : RDD

        RDD containing item pair frequencies.

    frequency\_x\_dict : dict

        A dictionary of item frequencies.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    RDD

        RDD containing item pairs with their corresponding confidence values.

    Raises

    ------

    Exception

        If any error occurs during the confidence calculation.

    Notes

    -----

    This function calculates the confidence of item pairs based on their antecedents.

    It uses a dictionary of item frequencies (frequency\_x\_dict) to compute the confidence

    as the ratio of the frequency of the item pair (x, y) to the frequency of item x.

    """

    try:

        confidence\_rdd = frequency\_xy\_rdd.map(lambda *x*: (

            x[0], (x[1], frequency\_x\_dict.get(x[0][0]), x[1] / frequency\_x\_dict.get(x[0][0]))))

        return confidence\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def top\_n\_item\_pair\_with\_confidence(*confidence\_rdd*, *n*, *logger*):

    """

    Get the top N item pairs with the highest occurrence count of item X, and occurrence count of item pair X and Y.

    Parameters

    ----------

    confidence\_rdd : RDD

        RDD containing item pairs with their confidence values.

    n : int

        The number of top item pairs to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    list

        A list of the top N item pairs with the highest occurrence count of item X, and occurrence count of item pair X and Y.

    Raises

    ------

    Exception

        If any error occurs during the retrieval of top item pairs.

    Notes

    -----

    This function sorts the item pairs in confidence\_rdd by occurrence count of item X,

    and occurrence count of item pair X and Y in descending order.

    """

    try:

        sorted\_items = confidence\_rdd.sortBy(

            lambda *x*: (x[1][1], x[1][0]), *ascending*=False).take(n)

        return sorted\_items

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def bottom\_n\_item\_pair\_with\_confidence(*confidence\_rdd*, *n*, *logger*):

    """

    Get the top N item pairs with the lowest occurrence count of item X, and occurrence count of item pair X and Y.

    Parameters

    ----------

    confidence\_rdd : RDD

        RDD containing item pairs with their confidence values.

    n : int

        The number of top item pairs to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    list

        A list of the top N item pairs with the lowest occurrence count of item X, and occurrence count of item pair X and Y.

    Raises

    ------

    Exception

        If any error occurs during the retrieval of top item pairs.

    Notes

    -----

    This function sorts the item pairs in confidence\_rdd by occurrence count of item X,

    and occurrence count of item pair X and Y in ascending order.

    """

    try:

        sorted\_items = confidence\_rdd.sortBy(

            lambda *x*: (x[1][1], x[1][0]), *ascending*=True).take(n)

        return sorted\_items

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def main():

    """Entry point of the script.

    Parameters

    ----------

    None

    Returns

    -------

    None

    Notes

    -----

    This function serves as the entry point of the script for processing flight data. It performs the following steps:

    1. Configures the logging settings and initializes a logger.

    2. Creates a SparkContext for data processing.

    3. Loads grocery data from a CSV file and cleanses it from trailing spaces.

    4. Displays and logs various statistics and analysis results, such as transaction counts, unique item counts, top and bottom items, support, etc.

    5. Computes confidence values for item pairs and displays the top and bottom item pairs with confidence.

    6. Stops the SparkContext when processing is complete.

    """

    logger = configure\_logging()

    sc = create\_spark\_context()

    try:

        grocery\_rdd = load\_data(sc, logger, GROCERY\_DATA\_FILE\_PATH)

        show\_rdd(grocery\_rdd, logger)

        occurrence = grocery\_rdd.count()

        logger.info(f"There are {occurrence} transactions in the rdd.\n")

        cleansed\_grocery\_rdd = cleanse(grocery\_rdd, logger)

        show\_rdd(cleansed\_grocery\_rdd, logger)

        occurrence = cleansed\_grocery\_rdd.count()

        logger.info(f"There are {occurrence} transactions in the rdd.\n")

        most\_groceries = transaction\_with\_most\_items(

            cleansed\_grocery\_rdd, logger)

        logger.info(most\_groceries)

        unique\_groceries\_count = count\_unique\_items(

            cleansed\_grocery\_rdd, logger)

        logger.info(

            f"There are {unique\_groceries\_count} unique items in the rdd.\n")

        top\_20\_items = top\_n\_item\_with\_percentage(

            cleansed\_grocery\_rdd, 20, logger)

        logger.info(top\_20\_items)

        bottom\_20\_items = bottom\_n\_item\_with\_percentage(

            cleansed\_grocery\_rdd, 20, logger)

        logger.info(bottom\_20\_items)

        indexed\_grocery\_rdd = add\_index(cleansed\_grocery\_rdd, logger)

        show\_rdd(indexed\_grocery\_rdd, logger)

        occurrence = indexed\_grocery\_rdd.count()

        logger.info(f"There are {occurrence} number of records.\n")

        combination\_2item = generate\_combinations(indexed\_grocery\_rdd, logger)

        show\_rdd(combination\_2item, logger)

        occurrence = combination\_2item.count()

        logger.info(f"There are {occurrence} number of records.\n")

        associated\_transaction\_combi = association(combination\_2item, logger)

        show\_rdd(associated\_transaction\_combi, logger)

        occurrence = associated\_transaction\_combi.count()

        logger.info(f"There are {occurrence} number of records.\n")

        sorted\_associated\_count = item\_pair\_counts(

            associated\_transaction\_combi, logger)

        show\_rdd(sorted\_associated\_count, logger)

        occurrence = sorted\_associated\_count.count()

        logger.info(f"There are {occurrence} number of records.\n")

        item\_pair\_support = support(sorted\_associated\_count, logger)

        show\_rdd(item\_pair\_support, logger)

        total\_records = sorted\_associated\_count.map(lambda *x*: x[1]).sum()

        logger.info(f"There are {total\_records} total number of records.\n")

        top\_20\_item\_pairs = top\_n\_item\_pair\_with\_support(

            item\_pair\_support, 20, logger)

        logger.info(top\_20\_item\_pairs)

        bottom\_20\_item\_pairs = bottom\_n\_item\_pair\_with\_support(

            item\_pair\_support, 20, logger)

        logger.info(bottom\_20\_item\_pairs)

        indexed\_grocery\_rdd = add\_index(cleansed\_grocery\_rdd, logger)

        show\_rdd(indexed\_grocery\_rdd, logger)

        occurrence = indexed\_grocery\_rdd.count()

        logger.info(f"There are {occurrence} number of records.\n")

        permutation\_2item = generate\_permutations(indexed\_grocery\_rdd, logger)

        show\_rdd(permutation\_2item, logger)

        occurrence = permutation\_2item.count()

        logger.info(f"There are {occurrence} number of records.\n")

        associated\_transaction\_permu = association(permutation\_2item, logger)

        show\_rdd(associated\_transaction\_permu, logger)

        occurrence = associated\_transaction\_permu.count()

        logger.info(f"There are {occurrence} number of records.\n")

        frequency\_xy = item\_pair\_counts(

            associated\_transaction\_permu, logger)

        show\_rdd(frequency\_xy, logger)

        occurrence = frequency\_xy.count()

        logger.info(f"There are {occurrence} number of records.\n")

        frequency\_x = item\_count(cleansed\_grocery\_rdd, logger)

        logger.info(frequency\_x)

        occurrence = len(frequency\_x)

        logger.info(f"There are {occurrence} number of records.\n")

        confidence\_rdd = confidence(frequency\_xy, frequency\_x, logger)

        show\_rdd(confidence\_rdd, logger)

        occurrence = confidence\_rdd.count()

        logger.info(f"There are {occurrence} number of records.\n")

        top\_20\_item\_pairs\_confidence = top\_n\_item\_pair\_with\_confidence(

            confidence\_rdd, 20, logger)

        logger.info(top\_20\_item\_pairs\_confidence)

        bottom\_20\_item\_pairs\_confidence = bottom\_n\_item\_pair\_with\_confidence(

            confidence\_rdd, 20, logger)

        logger.info(bottom\_20\_item\_pairs\_confidence)

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

    finally:

        if sc is not None:

            sc.stop()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**(a)**

***Script Constants Snippet***

# Constants

SCRIPTS\_DIR = os.path.abspath(os.path.dirname(\_\_file\_\_))

DATA\_DIR = os.path.join(SCRIPTS\_DIR, "..", "data")

GROCERY\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "grocery\_data.csv")

LOGGING\_LEVEL = logging.INFO

LOAD\_DATA\_ERROR\_MESSAGE = "An error occurred while loading data: {}"

FILE\_NOT\_FOUND\_MESSAGE = "The specified file does not exist: {}"

***Script Functions Snippet***

def show\_rdd(*rdd*, *logger*, *max\_rows*=100, *show\_rows*=20):

    if rdd.count() > max\_rows:

        logger.info(rdd.take(show\_rows))

    else:

        logger.info(rdd.collect())

def load\_data(*sc*, *logger*, *file\_path*):

    try:

        rdd = sc.textFile(file\_path)

        return rdd

    except *Exception* as e:

        if "Path does not exist" in *str*(e):

            logger.error(FILE\_NOT\_FOUND\_MESSAGE.format(file\_path))

            raise *FileNotFoundError*(f"File not found: {file\_path}")

        logger.error(LOAD\_DATA\_ERROR\_MESSAGE.format(*str*(e)))

        raise e

def cleanse(*rdd*, *logger*):

    try:

        cleansed\_rdd = rdd.map(

            lambda *x*: [item.strip().lower() for item in x.split(',')])

        return cleansed\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def transaction\_with\_most\_items(*cleanse\_rdd*, *logger*):

    try:

        transaction\_item\_count = cleanse\_rdd.map(

            lambda *items*: (items, len(items)))

        max\_transaction = transaction\_item\_count.max(*key*=lambda *x*: x[1])

        max\_transaction\_content, max\_transaction\_items = max\_transaction

        return max\_transaction\_content, max\_transaction\_items

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def count\_unique\_items(*cleanse\_rdd*, *logger*):

    try:

        unique\_items = cleanse\_rdd.flatMap(lambda *items*: items).distinct()

        count = unique\_items.count()

        return count

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        grocery\_rdd = load\_data(sc, logger, GROCERY\_DATA\_FILE\_PATH)

        show\_rdd(grocery\_rdd, logger)

        occurrence = grocery\_rdd.count()

        logger.info(f"There are {occurrence} transactions in the rdd.\n")

        cleansed\_grocery\_rdd = cleanse(grocery\_rdd, logger)

        show\_rdd(cleansed\_grocery\_rdd, logger)

        occurrence = cleansed\_grocery\_rdd.count()

        logger.info(f"There are {occurrence} transactions in the rdd.\n")

        most\_groceries = transaction\_with\_most\_items(

            cleansed\_grocery\_rdd, logger)

        logger.info(most\_groceries)

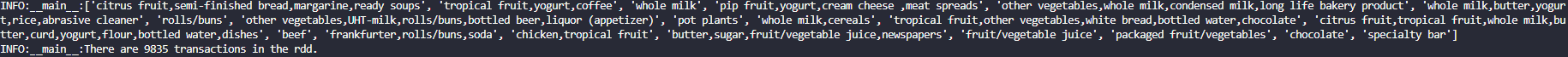
        unique\_groceries\_count = count\_unique\_items(

            cleansed\_grocery\_rdd, logger)

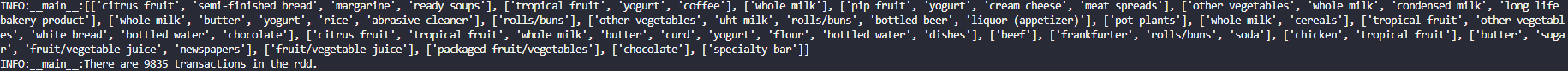
        logger.info(

            f"There are {unique\_groceries\_count} unique items in the rdd.\n")

***Output Screenshots***



*Figure 46. Content and number of transactions of grocery RDD.*



*Figure 47. Content and number of transactions of cleansed grocery RDD.*



*Figure 48. Content of transaction with the most number of items.*



*Figure 49. Number of unique items in the grocery data.*

**(b)**

***Script Functions Snippet***

def top\_n\_item\_with\_percentage(*cleanse\_rdd*, *n*, *logger*):

    try:

        item = cleanse\_rdd.flatMap(lambda *items*: items)

        item\_count = item.countByValue().items()

        sorted\_item\_count = sorted(

            item\_count, *key*=(lambda *x*: x[1]), *reverse*=True)

        top\_item = sorted\_item\_count[:n]

        total\_transaction = item.count()

        top\_items\_with\_percentage = [

            (item, count, ((count / total\_transaction) \* 100)) for item, count in top\_item]

        return top\_items\_with\_percentage

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def bottom\_n\_item\_with\_percentage(*cleanse\_rdd*, *n*, *logger*):

    try:

        item = cleanse\_rdd.flatMap(lambda *items*: items)

        item\_count = item.countByValue().items()

        sorted\_item\_count = sorted(

            item\_count, *key*=(lambda *x*: x[1]), *reverse*=False)

        bottom\_item = sorted\_item\_count[:n]

        total\_transaction = item.count()

        bottom\_items\_with\_percentage = [

            (item, count, ((count / total\_transaction) \* 100)) for item, count in bottom\_item]

        return bottom\_items\_with\_percentage

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        top\_20\_items = top\_n\_item\_with\_percentage(

            cleansed\_grocery\_rdd, 20, logger)

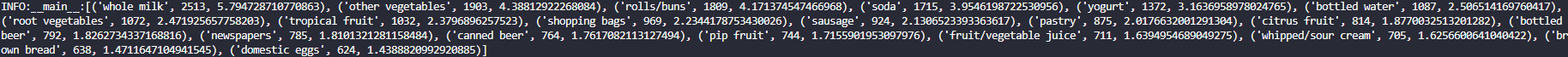
        logger.info(top\_20\_items)

        bottom\_20\_items = bottom\_n\_item\_with\_percentage(

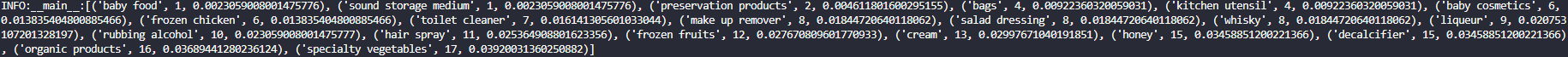
            cleansed\_grocery\_rdd, 20, logger)

        logger.info(bottom\_20\_items)

***Output Screenshots***



*Figure 51. Sorted top 20 items of cleansed grocery RDD.*



*Figure 52. Sorted bottom 20 items of cleansed grocery RDD.*

**(c)**

***Script Functions Snippet***

def show\_rdd(*rdd*, *logger*, *max\_rows*=100, *show\_rows*=20):

    if rdd.count() > max\_rows:

        logger.info(rdd.take(show\_rows))

    else:

        logger.info(rdd.collect())

def add\_index(*cleanse\_rdd*, *logger*):

    try:

        indexed\_rdd = cleanse\_rdd.zipWithIndex()

        return indexed\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def generate\_combinations(*indexed\_rdd*, *logger*):

    try:

        item\_combinations\_rdd = indexed\_rdd.flatMap(lambda *transaction*: [(

            (item1, item2), transaction[1])

            for item1 in transaction[0] for item2 in transaction[0]

            # Ensure item1 is less than item2 (in alphabetical order) to avoid duplicate pairs

            if item1 < item2

        ])

        sorted\_item\_combinations\_rdd = item\_combinations\_rdd.sortBy(

            lambda *x*: (x[1], x[0]))

        return sorted\_item\_combinations\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def association(*combinations\_rdd*, *logger*):

    try:

        transaction\_indices = combinations\_rdd.groupByKey().map(

            lambda *x*: (x[0], *list*(x[-1])))

        return transaction\_indices

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def item\_pair\_counts(*association\_rdd*, *logger*):

    try:

        item\_count\_rdd = association\_rdd.map(lambda *x*: (x[0], len(x[1])))

        sorted\_item\_count\_rdd = item\_count\_rdd.sortBy(

            lambda *x*: x[1], *ascending*=False)

        return sorted\_item\_count\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        indexed\_grocery\_rdd = add\_index(cleansed\_grocery\_rdd, logger)

        show\_rdd(indexed\_grocery\_rdd, logger)

        occurrence = indexed\_grocery\_rdd.count()

        logger.info(f"There are {occurrence} number of records.\n")

        combination\_2item = generate\_combinations(indexed\_grocery\_rdd, logger)

        show\_rdd(combination\_2item, logger)

        occurrence = combination\_2item.count()

        logger.info(f"There are {occurrence} number of records.\n")

        associated\_transaction = association(combination\_2item, logger)

        show\_rdd(associated\_transaction, logger)

        occurrence = associated\_transaction.count()

        logger.info(f"There are {occurrence} number of records.\n")

        sorted\_associated\_count = item\_pair\_counts(

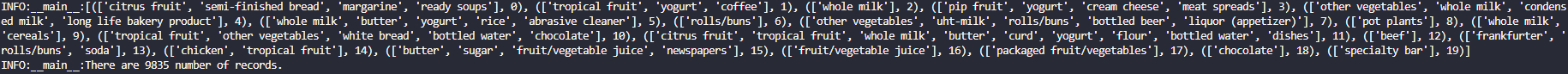
            associated\_transaction, logger)

        show\_rdd(sorted\_associated\_count, logger)

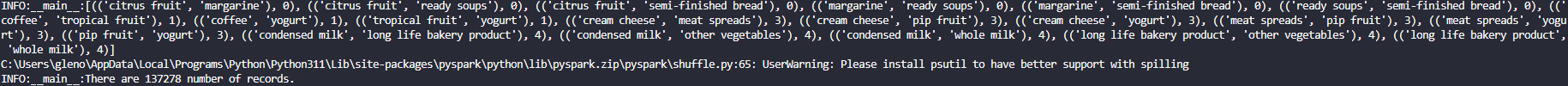
        occurrence = sorted\_associated\_count.count()

        logger.info(f"There are {occurrence} number of records.\n")

***Output Screenshots***



*Figure 53. Content and number of indexed transactions.*

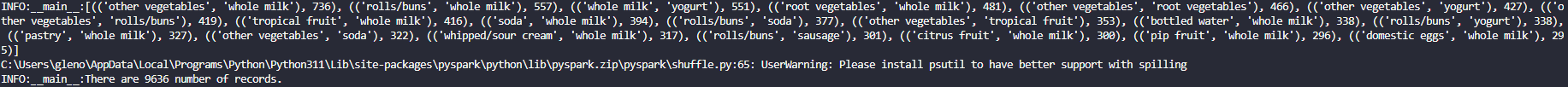


*Figure 54. Sorted content and number of the 2 itemset combination within each transaction.*

A screen shot of a computer screen

Description automatically generated

*Figure 55. Content and number of the itemset and the associated transaction indices.*



*Figure 56. Content and number of the itemset and the count of associated transaction indices.*

**(d)**

***Script Functions Snippet***

def support(*item\_pair\_counts\_rdd*, *logger*):

    try:

        total\_records = item\_pair\_counts\_rdd.map(lambda *x*: x[1]).sum()

        support\_rdd = item\_pair\_counts\_rdd.map(lambda *x*: (

            (x[0], (x[1], x[1] / total\_records))))

        return support\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def top\_n\_item\_pair\_with\_support(*support\_rdd*, *n*, *logger*):

    try:

        sorted\_items = support\_rdd.sortBy(

            lambda *x*: x[1][0], *ascending*=False).take(n)

        return sorted\_items

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def bottom\_n\_item\_pair\_with\_support(*support\_rdd*, *n*, *logger*):

    try:

        sorted\_items = support\_rdd.sortBy(

            lambda *x*: x[1][0], *ascending*=True).take(n)

        return sorted\_items

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        item\_pair\_support = support(sorted\_associated\_count, logger)

        show\_rdd(item\_pair\_support, logger)

        total\_records = sorted\_associated\_count.map(lambda *x*: x[1]).sum()

        logger.info(f"There are {total\_records} total number of records.\n")

        top\_20\_item\_pairs = top\_n\_item\_pair\_with\_support(

            item\_pair\_support, 20, logger)

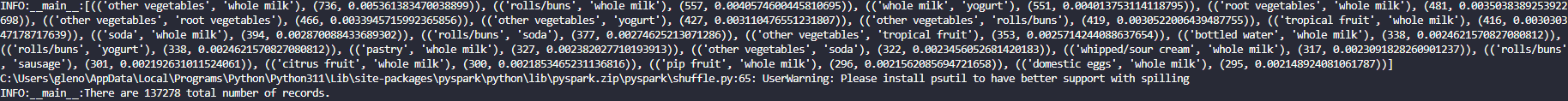
        logger.info(top\_20\_item\_pairs)

        bottom\_20\_item\_pairs = bottom\_n\_item\_pair\_with\_support(

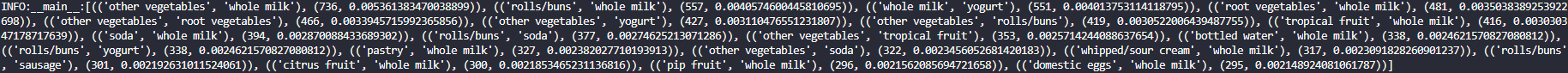
            item\_pair\_support, 20, logger)

        logger.info(bottom\_20\_item\_pairs)

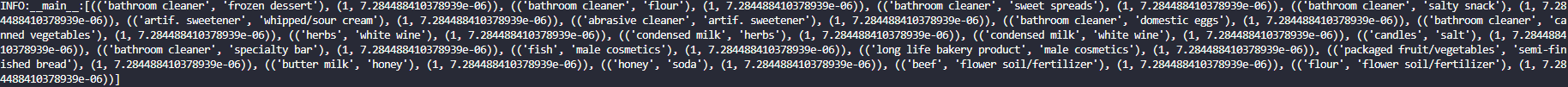
***Output Screenshots***



*Figure 57. Content and number of the itemset count and support.*



*Figure 58. Sorted top 20 items of itemset with count and support.*

**

*Figure 59. Sorted bottom 20 items of itemset with count and support.*

**(e)**

***Script Functions Snippet***

def show\_rdd(*rdd*, *logger*, *max\_rows*=100, *show\_rows*=20):

    if rdd.count() > max\_rows:

        logger.info(rdd.take(show\_rows))

    else:

        logger.info(rdd.collect())

def add\_index(*cleanse\_rdd*, *logger*):

    try:

        indexed\_rdd = cleanse\_rdd.zipWithIndex()

        return indexed\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def association(*combinations\_rdd*, *logger*):

    try:

        transaction\_indices = combinations\_rdd.groupByKey().map(

            lambda *x*: (x[0], *list*(x[-1])))

        return transaction\_indices

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def item\_pair\_counts(*association\_rdd*, *logger*):

    try:

        item\_count\_rdd = association\_rdd.map(lambda *x*: (x[0], len(x[1])))

        sorted\_item\_count\_rdd = item\_count\_rdd.sortBy(

            lambda *x*: x[1], *ascending*=False)

def generate\_permutations(*indexed\_rdd*, *logger*):

    try:

        def generate\_pairs(*transaction*):

            item\_list, transaction\_index = transaction

            item\_pairs = []

            # Iterate through items in the transaction

            for i in range(len(item\_list)):

                for j in range(i + 1, len(item\_list)):

                    item1 = item\_list[i]

                    item2 = item\_list[j]

                    # Create pairs for item1 and item2, as well as their reverse order

                    item\_pairs.append(((item1, item2), transaction\_index))

                    item\_pairs.append(((item2, item1), transaction\_index))

            return item\_pairs

        permutations\_rdd = indexed\_rdd.flatMap(generate\_pairs)

        sorted\_permutations\_rdd = permutations\_rdd.sortBy(

            lambda *x*: (x[1], x[0]))

        return sorted\_permutations\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def item\_count(*cleansed\_rdd*, *logger*):

    try:

        # Transform each transaction into a list of (item, 1) pairs, then reduce to count occurrences.

        item\_frequencies\_rdd = cleansed\_rdd.flatMap(lambda *transaction*: [(

            item, 1) for item in transaction]).reduceByKey(lambda *a*, *b*: a + b)

        sorted\_item\_counts\_rdd = item\_frequencies\_rdd.sortBy(

            lambda *x*: x[1], *ascending*=False)

        sorted\_item\_counts\_dict = *dict*(sorted\_item\_counts\_rdd.collect())

        return sorted\_item\_counts\_dict

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        indexed\_grocery\_rdd = add\_index(cleansed\_grocery\_rdd, logger)

        show\_rdd(indexed\_grocery\_rdd, logger)

        occurrence = indexed\_grocery\_rdd.count()

        logger.info(f"There are {occurrence} number of records.\n")

        permutation\_2item = generate\_permutations(indexed\_grocery\_rdd, logger)

        show\_rdd(permutation\_2item, logger)

        occurrence = permutation\_2item.count()

        logger.info(f"There are {occurrence} number of records.\n")

        associated\_transaction\_permu = association(permutation\_2item, logger)

        show\_rdd(associated\_transaction\_permu, logger)

        occurrence = associated\_transaction\_permu.count()

        logger.info(f"There are {occurrence} number of records.\n")

        frequency\_xy = item\_pair\_counts(

            associated\_transaction\_permu, logger)

        show\_rdd(frequency\_xy, logger)

        occurrence = frequency\_xy.count()

        logger.info(f"There are {occurrence} number of records.\n")

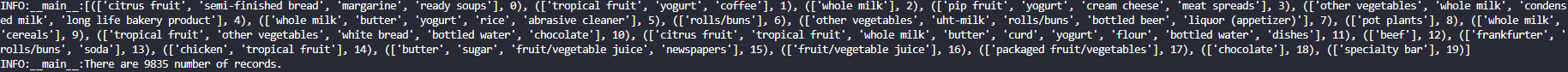
        frequency\_x = item\_count(cleansed\_grocery\_rdd, logger)

        logger.info(frequency\_x)

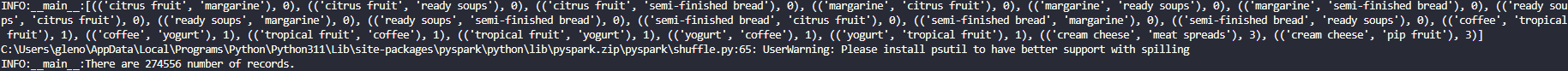
        occurrence = len(frequency\_x)

        logger.info(f"There are {occurrence} number of records.\n")

***Output Screenshots***



*Figure 60. Content and number of indexed transactions.*

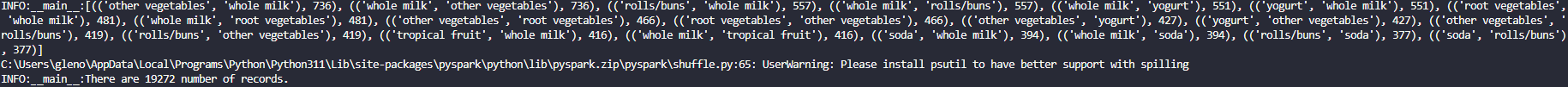


*Figure 61. Content and number of the sorted 2 itemset sorted permutation within each transaction*

A screen shot of a computer screen

Description automatically generated

*Figure 62. Content and number of the itemset and the associated transaction indices.*

**

*Figure 63. Sorted content and number of the itemset and the count of associated transaction indices.*

A blurry image of a computer screen

Description automatically generated

*Figure 64. Sorted content and number of the item count and support.*

**(f)**

***Script Functions Snippet***

def show\_rdd(*rdd*, *logger*, *max\_rows*=100, *show\_rows*=20):

    if rdd.count() > max\_rows:

        logger.info(rdd.take(show\_rows))

    else:

        logger.info(rdd.collect())

def confidence(*frequency\_xy\_rdd*, *frequency\_x\_dict*, *logger*):

    try:

        confidence\_rdd = frequency\_xy\_rdd.map(lambda *x*: (

            x[0], (x[1], frequency\_x\_dict.get(x[0][0]), x[1] / frequency\_x\_dict.get(x[0][0]))))

        return confidence\_rdd

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def top\_n\_item\_pair\_with\_confidence(*confidence\_rdd*, *n*, *logger*):

    try:

        sorted\_items = confidence\_rdd.sortBy(

            lambda *x*: (x[1][1], x[1][0]), *ascending*=False).take(n)

        return sorted\_items

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def bottom\_n\_item\_pair\_with\_confidence(*confidence\_rdd*, *n*, *logger*):

    try:

        sorted\_items = confidence\_rdd.sortBy(

            lambda *x*: (x[1][1], x[1][0]), *ascending*=True).take(n)

        return sorted\_items

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

***Script Main Function Snippet***

        confidence\_rdd = confidence(frequency\_xy, frequency\_x, logger)

        show\_rdd(confidence\_rdd, logger)

        occurrence = confidence\_rdd.count()

        logger.info(f"There are {occurrence} number of records.\n")

        top\_20\_item\_pairs\_confidence = top\_n\_item\_pair\_with\_confidence(

            confidence\_rdd, 20, logger)

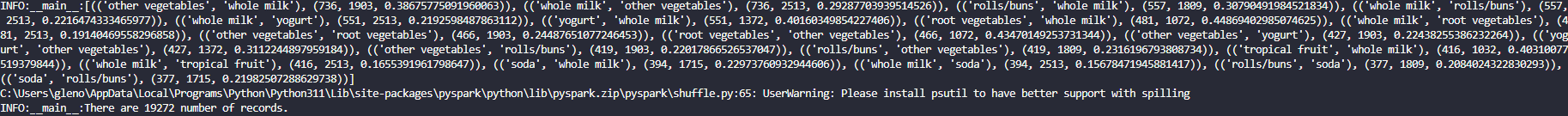
        logger.info(top\_20\_item\_pairs\_confidence)

        bottom\_20\_item\_pairs\_confidence = bottom\_n\_item\_pair\_with\_confidence(

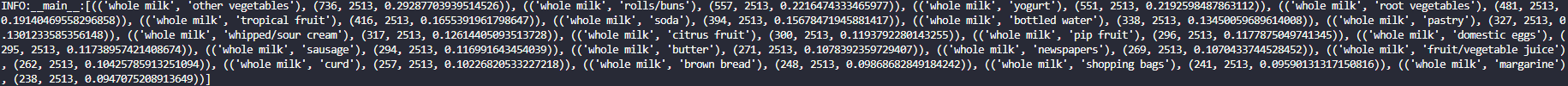
            confidence\_rdd, 20, logger)

        logger.info(bottom\_20\_item\_pairs\_confidence)

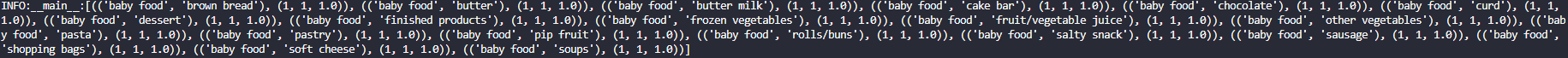
***Output Screenshots***



*Figure 65. Content and number of the itemset count, item count and confidence.*



*Figure 66. Sorted top 20 items of itemset with count, item count and confidence.*



*Figure 66. Sorted bottom 20 items of itemset with count, item count and confidence.*